Chapter 8

Hardware Virtualization

Topics covered in this chapter include:

- Applications of virtualization, including server consolidation
- Software emulation, full binary translation, and classical virtualization
- Kernel binary translation, hardware virtualization, and paravirtualization
- Virtual machine migration
- Hosted vs. "bare-metal" hypervisors
- Containers and Docker (even though they don't use HW virtualization)

Hardware virtualization is a technique that allows multiple virtual machines (VMs) to run on the same physical machine, using either pure software or a combination of hardware and software techniques.

Previous chapters have described the differences between *threads*—separate flows of control sharing (almost) all resources such as memory and file descriptors—and *processes*, which are isolated from each other by the operating system, requiring the use of files, pipes, or similar mechanisms to communicate between two processes. A virtual machine is similar to a process, but is designed to run a full operating system and its applications, rather than a single program; communication between VMs is like that between real machines, and must take place over (possibly emulated) networks.

A virtual machine requires a much different interface—while a process runs in unprivileged mode, performing I/O and memory management operations by issuing system calls to the OS kernel, an operating system runs in supervisor mode and uses special instructions and other hardware

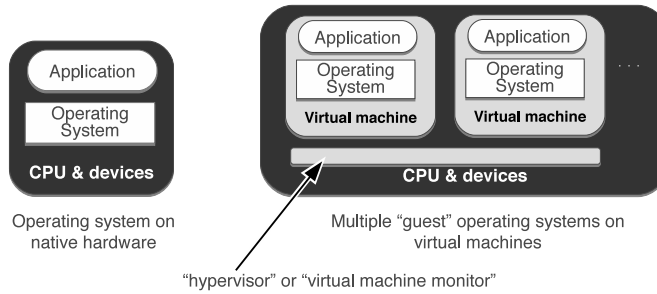

Figure 8.1: Virtual machine architecture

mechanisms to perform its operations. In a virtual machine, as shown in Figure 8.1, these mechanisms are performed by the *hypervisor*[1] which sits "underneath" the operating system.

Running multiple operating system instances on the same physical hardware serves a number of purposes:

Running multiple operating systems: Many applications are tied to a specific OS or even OS version; by using virtual machines it is possible to run instances of these other operating systems and make these applications availabe to a user without requiring extra hardware. (As an example, the laptop I am typing on runs Apple's OS X, but I have a virtual machine running Ubuntu Linux for Linux development.)

Multiple Configurations: Even applications which run on the same operating system may need to run on different machines, rather than just in separate processes. This may be because they require different, incompatible versions of system libraries, or different configuration options. In some cases (e.g. running an old and new version of the same application) they may need different versions of the same configuration files.

Supporting multiple configurations is frequently called *server consolidation*, as in the past an enterprise may have needed to use multiple physical machines to provide these configurations. Frequently the load on each service or configuration was much less than what could be handled by a single machine, and many of these services can instead be deployed as virtual machines on a single physical system.

Security: Many applications (e.g. webservers, databases) require administrative privileges (e.g. root on Unix) for configuration. In the past these applications were typically considered infrastructure services, maintained

[1] Early operating systems were often called *supervisors*; what do you call the program which supervises the supervisor? A hypervisor, I guess.

and configured by system administrators at the request of users. However in many recent cases (e.g. Amazon's Elastic Compute Cloud) the customer is expected to perform all configuration and management, and multiple untrusted customers may share the same physical hardware. Instead of being provided an unprivileged login on a shared machine, each customer is given a virtual machine which they can configure as they wish, with full root or administrative privileges, without posing a threat to customers on other virtual machines.

These uses for virtual machines are artifacts of how applications and operating systems have evolved, and a perfectly-designed OS would no doubt provide the security and manageability benefits described above using operating system-level protections. (This would of course eliminate the need to use any other less perfect operating system.) Virtual machines hold another security advantage, however: they have a smaller *attack surface* than general-purpose operating systems. Operating systems are very large, with millions of lines of code.[2] A *hypervisor*, the piece of software responsible for managing virtual machines, is typically far smaller in comparison, and has only a small number of external interfaces. In theory fewer lines of code (espcially the security-critical code which validates user inputs) means fewer bugs and thus fewer opportunities for security exploits; experience to date seems to support this theory.

> Operating system *containers*, such as those used by Docker, provide many of the advantages of virtual machines while using a single operating system. Each container is a set of processes with a *namespace* of process IDs and network connections, and a separate file system tree, and (barring misconfiguration or kernel bugs) is unable to access resources belonging to other containers or to the host OS.

Review Questions

8.0.1. Which of these are reasons why it can be difficult to run multiple network servers on the same machine with a normal operating system?

a) Problems related to assigning separate network addresses to different servers
b) Conflicts between the software and OS requirements for different software packages

[2]The `kernel/` and `mm/` directories in the Linux source add up to about a third of a million lines of code; support for Intel CPUs in `arch/x86/` is another third of a million; the `drivers/` directory is over ten million LOC.

c) The need for administrative privileges to install software packages
d) All of the above

8.1 Implementing Hardware Virtualization

If you are used to running VirtualBox or VMware on your laptop, it may seem like it's just another program, maybe using more memory and CPU than most. But it isn't. To understand why, consider trying to run Linux (the "guest" operating system) on top of a "host" operating system, e.g. Windows. The linux kernel is an executable file, typically found in `/boot/vmlinux`, and could be readily translated into a Windows executable. However if you tried to do this[3] it would crash immediately. Some of the reasons an operating system kernel cannot run as a process are:

Privileged instructions: One of the first things the kernel does on startup is to initialize the virtual memory system, mapping virtual addresses to physical addresses. This configuration requires privileged-mode instructions, which are inaccessible to user-mode applications, as they could be used to bypass operating system protections. The first such instruction executed by the guest OS would cause an exception, killing the process.

Interference: The problem isn't just that the guest OS won't be allowed to modify virtual address mappings. If it actually could modify these mappings, then the underlying host operating system would almost certainly crash, as it assumes that it has complete control over them. The CPU only has a single address translation mechanism, and if two operating systems are going to make use of it, they must either deliberately share access, or it must be virtualized before being used by one or both OS.

Security: Secure isolation between virtual machines, including memory protection, is at least as important as isolation between processes in a normal operating system. But if a guest operating system has direct access to the CPU address translation mechanisms it can easily access physical memory allocated to another virtual machine (or to the host OS itself), bypassing any security mechanisms.

I/O: A process running under Linux or Windows uses *system calls* such as **open** and **read** to access *files*. In contrast, an operating system uses *drivers* to access *physical devices*.

[3] or, actually, running any OS on top of any OS including itself

```
char memory[EMULATED_MEM_SIZE];
int R1, R2, R3, ...;
int PC, SP, CR1, CR2, CR3, ...;
bool S; /* supervisor mode */

while (true):
   instr = memory[PC]
   PC += sizeof(instr)
   case (instr) in:
     "MOV R1 -> R2":
       R2 = R1
     case "JMP <arg>":
       PC = <arg>
     case "STORE Rx, <addr>":
          <paddr> = MMU_translate[<addr>]
                  - on error: emulate page fault
       if <paddr> is real memory:
         memory[paddr] = Rx
       else
         simulate_IO_access(Write, paddr, Rx)
      .... Etc. (for ~1000 more instructions)
```

Figure 8.2: Hypothetical software emulation

In the remainder of this chapter we discuss the following approaches to supporting virtual machines, arranged (roughly) in increasing order of both complexity and performance:

- Software emulation.
- Emulation with binary translation.
- Classical (direct execution + trap-and-emulate) virtualization
- Direct execution + binary translation
- Hardware-assisted virtualization
- Paravirtualization.

Software emulation

The most straightforward way to run a virtual machine is to emulate it entirely in software: in other words, to write a program that behaves exactly like the CPU, memory, and I/O devices of the real machine. The idea is simple: given a complete description of how the CPU behaves, create variables for the registers and a big array for memory, and write a program that repeatedly fetches instructions from the memory array, decodes them, and emulates their operation, much like the sample code in Figure 8.2.

Full software emulation is simple conceptually, although in practice the list of instructions to implement can get long (over 1000 on modern x86 CPUs) and complex. It has one major advantage, portability: once the code to emulate a specific CPU is written, it can be compiled and run on almost any host. This is especially useful in embedded development, where it is often necessary to develop and test software before the CPU (or at least the system incorporating that CPU) is ready to use.

> The Java Virtual Machine (JVM) executes bytecode instructions, and can be considered a sort of CPU. Almost all JVMs are based on software emulation, typically with additional performance optimizations.

The primary disadvantage is performance—full software emulation is slow. It can be hundreds of times slower than native execution, making it unsuited for all but a few applications.

Emulation plus binary translation

Software interpreters can be sped up by what Java developers call Just In Time (JIT) compilation, and which CPU emulator developers call Binary Translation. The idea is to translate commonly-used fragments of code into actual machine code, which can usually run far faster than pure emulation. (In part, it eliminates the software-implemented instruction fetch and decode for each instruction, which is a significant overhead.) An example can be seen in Figure 8.3.

> Other software systems which use binary translation techniques include:
> **JVMs:** Almost all Java implementations use JIT compiling for performance.
> **Javascript:** Recent browsers (Safari, Firefox, and others) use just-in-time compilation to improve Javascript performance.
> **Apple Rosetta:** This allowed Intel-based Macintosh computers to execute programs compiled for PowerPC.

In other words, the following occurs when a section of binary-translated code is executed:

1. The real CPU registers are loaded from the virtual (i.e., software-emulated) registers
2. The translated instructions are executed
3. The virtual CPU state is updated with results from the real CPU

In most cases the translator will produce more than one instruction per emulated instruction. Memory accesses are particularly tricky, as the generated code must emulate address translation performed by the MMU,

The following code:

```
ADD  R1+R2 -> R2
ADD  R2+R3 -> R3
MUL  2,R3  -> R3
```

might be translated into the following fragment:

```
LOAD Rx <- &emulated_R1
LOAD Ry <- &emulated_R2
LOAD Rz <- &emulated_R3
ADD Rx,Ry -> Ry
ADD Ry,Rz -> Rz
MUL 2,Rz -> Rz
STORE Ry -> &emulated_R2
STORE Rz -> &emulated_R3
RET
```

Figure 8.3: Example of binary translation

and then check to see whether the resulting address is I/O or RAM before performing the operation. In practice it may be possible to arrange emulator memory (using e.g. the `mmap` system call) so that most memory accesses can be performed directly; in this case the overhead for most memory accesses can be reduced to a few instructions which check that an access falls within this typical range.

Trap and Emulate

Even with binary translation, software emulation is still slow—even the best binary translation systems incur a slowdown of 3x to 10x compared to running directly on the same hardware. This is much better than unoptimized software emulation, and may be the best that can be done when emulating one CPU on top of a CPU running a different instruction set. (e.g. running iPhone or Android applications on an Intel-based laptop or desktop, or the Rosetta emulator which Apple used to allow PowerPC applications to run seamlessly on early Intel-based Macintosh systems.)

However In many other cases—such as VirtualBox running on my laptop—the CPU emulated by the virtual machine is the same as the real, physical CPU. In this case we can improve performance greatly by using *direct execution* when possible: executing instructions directly on the physical CPU. The only reason we were emulating instructions in the first place was because the host OS could not allow a virtual machine to directly execute certain privileged instructions, so the goal here is to emulate only these privileged instructions while directly executing all others.

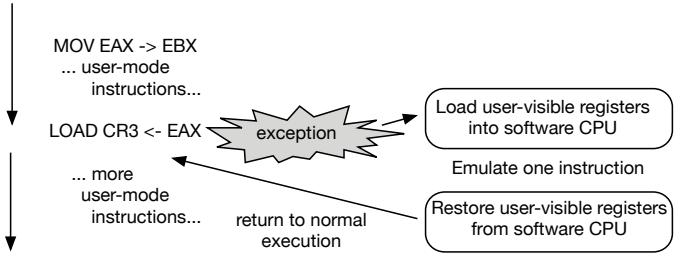

Figure 8.4: Trap-and-emulate virtualization

This can be done[4] using a strategy that can be called *trap-and-emulate*. The guest OS is executed directly in user mode; when it executes a privileged instruction it causes an exception which is handled by a specialized OS called a *hypervisor* or *virtual machine monitor*. The hypervisor loads the user-visible CPU registers into the software CPU emulation, runs it for a single instruction, and then returns back to direct execution.

It is interesting to compare a hypervisor running a guest OS (and guest applications) with a traditional operating system running multiple processes. A normal OS virtualizes CPU, memory, and other resources to provide a virtual machine abstraction to each process: each process sees its own memory space and a CPU which can execute user-mode instructions and a special system call instruction. A hypervisor, in contrast, performs a similar task of virtualizing memory and CPU, but provides a virtual machine abstraction which is identical to that of the hardware itself.

Figure 8.4 shows a representation of this trap-and-emulate process. It allows almost all instructions to run directly, at native hardware speed, while the specific instructions which need to be executed in privileged mode (a tiny fraction of all instructions) are emulated. This form of virtualization was originally developed by IBM in the late 60s and early 70s for mainframes, where it continues to be used.

But how does a hypervisor handle exceptions? An operating system relies heavily on exceptions; in fact, just about everything an OS does is part of some exception handler, whether that exception is a system call, a page fault, or a timer or I/O device interrupt. Since the guest OS runs in user mode, exceptions such as system calls or page faults generated by guest applications will be delivered to the hypervisor rather than the guest OS. The solution is for the hypervisor to just emulate the real CPU operation:

[4] on the right processors, as described below

1. Set the emulated supervisor bit to 1
2. (with the emulated CPU) Handle user/supervisor stack switch, pushing registers, and all the other exception-handling mechanisms that take up so many pages in the CPU reference manuals.
3. Return to user mode, load user-visible registers from the emulated CPU, and call the guest OS exception handler.
4. When the guest exception handler returns, set the emulated supervisor bit to 0, restore user registers from the kernel stack, switch to user stack, then jump back to direct execution at the instruction where the exception occurred[5].

How does it know where to find that exception handler? The hardware CPU locates exception handlers via one or more control registers which point to interrupt descriptors which are located in memory. (e.g in Intel-compatible CPUS the `IDTR` register, which points to the *interrupt descriptor table*) These registers may only be accessed in supervisor mode, so the hypervisor is able to virtualize access to these registers and maintain a separate emulated copy for each virtual machine. The real hardware register points to the hypervisor exception handler table, and when a hypervisor exception handler determines that an exception should be forwarded to the guest operating system it looks in the table pointed to by the emulated register to find the guest OS exception handler.

Review questions

8.1.1. Which of the following statements are true?

a) Software emulation uses special-purpose CPU hardware to run virtual machine instructions.
b) Software emulation is slower than native execution.

8.1.2. Trap-and-emulate virtualization:

a) Allows the guest OS to run in user mode, and intercepts exceptions that occur when it executes privileged instructions
b) Allows the guest OS to run in supervisor mode, and intercepts exceptions that occur when it executes privileged instructions
c) Prevents exceptions from occurring while the guest OS is executing

[5]Or the immediately following instruction in the case of *traps* such as system calls.

8.2 Virtualized memory

In a full software emulation, guest virtual addresses were translated into accesses to "fake" physical memory, e.g. the `memory[ ]` array in the example code. However, with trap-and-emulate virtualization, guest applications and most OS code execute directly on the CPU, and virtual addresses are translated to physical addresses in hardware, by the TLB and page tables. This is a problem, because to run multiple virtual machines on a single host, the hypervisor must be able to prevent each from accessing physical memory assigned to the other. Further complicating things, in many cases each guest OS expects physical memory to be in the same place, typically starting at physical address 0. This requires two levels of address translation to get from virtual addresses (used by the guest applications and OS) to real physical memory:

1. Virtual address to "fake" physical address: this translation is maintained by the guest OS
2. "Fake" physical address to real physical address: this translation is maintained by the hypervisor

How does this work on a CPU that only supports one level of virtual-to-physical address translation? By having the hypervisor maintain the real page tables (the ones pointed to by the real CR3) and making sure these tables contain the full virtual → fake physical → real physical translation. This requires two page tables: one pointed to by the emulated CR3 and used by the guest, and one "shadow" table that the real CR3 points to. When a page fault occurs the hypervisor page fault handler uses the following logic:

```
If faulting address is in guest page table:
  1. Look up virtual-to-fake-physical (guest page table) and
       fake physical-to-real-physical (hypervisor) mappings
  2. Install virtual → real physical mapping in
       shadow page table
  3. Return
else (not in guest page table):
  1. invoke guest OS page fault handler
```

Review questions

8.2.1. Address translation for a virtual machine is handled by:

a) Allowing the guest OS to maintain control of the hardware page tables.
b) Having the hypervisor determine the mappings which go in the hardware page tables

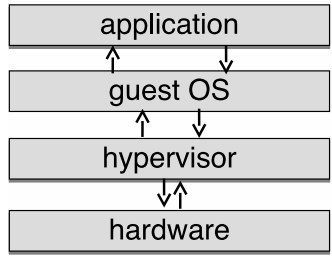

Figure 8.5: "Type 1" Hypervisor — no host OS.

c) Loading the hardware page tables with mappings which combine the guest OS page tables and the hypervisor memory maps.

8.3 Virtualized I/O Devices

Memory-mapped I/O devices are straightforward to emulate in a trap-and-emulate system. When the guest OS maps the physical memory addresses of emulated device registers, the hypervisor leaves the corresponding pages unmapped in the shadow table, so that all read and write accesses will result in a page fault. The hypervisor page fault handler handles faults on these pages specially, calling code that emulates reading from or writing to the emulated I/O device.

8.4 Hosted and "bare-metal" hypervisors

In Figure 8.5 you can see how this works together. Exceptions from user applications (page faults, system calls, etc.) are handled by the hypervisor, which in most cases passes them to the guest OS. Interrupts from I/O devices are passed to drivers in the hypervisor, which may in turn decide that it's time for a virtual hardware device to send an interrupt to a guest OS.

This image describes server systems (like VMware ESX), where the machine boots the hypervisor instead of a regular OS, and does nothing but run virtual machines. But what about a "hosted" system like VirtualBox or VMware Workstation? In particular, how does it run "on top of" a host OS?

The short answer is, it doesn't. When you install VirtualBox it installs a set of drivers, which (like normal hardware drivers) run as part of the kernel, in supervisor mode. When a virtual machine starts running, these

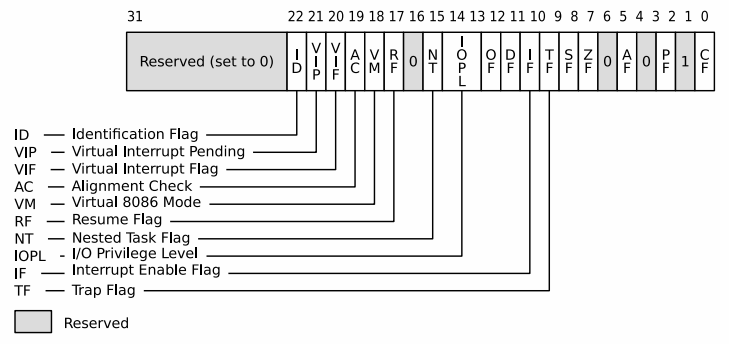

Figure 8.6: Sensitive state in the Intel architecture EFLAGS register

drivers insert themselves under the host operating system, "stealing" exceptions such as page faults and system calls whenever a virtual machine is executing, and forwarding them to the hypervisor. Running on the same system as a host operating system has its advantages, though, as the host OS has drivers for all of its hardware, a file system, display, and other useful interfaces. A hosted hypervisor can take advantage of this, passing I/O requests back to the host OS (via a rather complicated route) to be handled through these standard interfaces, rather than requiring its own drivers for any hardware it uses[6].

8.5 Non-Virtualizable CPUs

There is a minor problem with the classic trap-and-emulate virtualization mechanism as described above: it doesn't work. Or rather, it doesn't work on the machines you want it to work on.

In order to perform classic virtualization to work, every "sensitive" instruction (in other words, one that has to be emulated, like loading CR3 to switch page tables) must trap so that it can be emulated by the hypervisor. Unfortunately, some CPU architectures (in particular, Intel x86 CPUs) have instructions that fail this requirement. For example, on x86 CPUs, a number of instructions which modify supervisor-mode state will silently do nothing when executed in user mode, rather than causing an exception.

As an example, the EFLAGS register as shown in Figure 8.6 contains some commonly-used flags such as carry (CF) and zero (ZF) which it inherited

[6]That's how it works with binary emulation. With hardware virtualization support, the CPU has provisions to allow the "root" environment to continue to run without virtualization, but it's complicated.

from the 16-bit 8086, as well as system flags such as interrupt enable and "IO privilege level", the CPU user/supervisor privilege level. The POPF instruction modifies this register, by loading it with a value popped from the top of the stack. To prevent application code from arbitrarily disabling interrupts or turning on supervisor mode, when POPF is executed in user mode it silently ignores any privileged bits like IOPL and interrupt enable; when kernel code executes POPF in supervisor mode, these bits are loaded with new values. If we try to run the kernel in user mode this instruction will silently do the wrong thing, rather than trapping into the hypervisor.

Instructions like this complicate the task of performing efficient virtualization, but it is still possible, using one of three approaches:

- **Emulation with binary translation:** This is the simplest approach to describe, although rather difficult to implement well. Whenever the guest transitions into supervisor mode (for example, for a system call or an interrupt) the hypervisor emulates all instructions in software, using binary to translation speed up this process, and only resuming direct execution when the guest returns to user mode. This is slower for normal instructions in the kernel, but faster for privileged instructions, as it can translate them once instead of incurring the overhead of trapping and emulating each privileged instruction every time it executes.
- **Hardware virtualization:** Modern x86 CPUs include virtualization extensions, which add a third privilege level more powerful than supervisor mode. The guest runs in normal user and supervisor mode, but certain instructions trap into hypervisor mode for emulation, just as in trap-and-emulate virtualization on a virtualizable CPU. Which instructions? It depends: there are configuration registers providing the hypervisor with a menu of which operations it wants to intercept.
- **Paravirtualization:** rather than providing complete emulation of the hardware platform, the hypervisor provides an OS-like interface so that the operating system can request operations (e.g. address space switch) which would be performed by hardware instructions (e.g. LOAD CR3) on bare hardware. The guest operating system must be modified to use these requests, and so while binary translation and hardware virtualization can run unmodified guest operating systems (e.g. standard Windows installation media) paravirtualization can only support guest operating systems which have been modified for paravirtualization.
  The changes required in a guest OS are actually not that extensive, as most modern operating systems (even Windows) are structured so that they can be (relatively) easily modified to support different

machine types, with hardware-specific portions isolated into a small, replaceable part of the code.

A paravirtualized hypervisor looks sort of like a regular OS: it runs in supervisor mode, with guests running in user mode making requests via system calls using TRAP instructions. Unlike a normal OS, however, these system calls perform hardware-level operations like loading a page table, allocating physical memory, or installing a page fault handler.

Although paravirtualization requires some modifications to the guest OS, in some cases it provides higher performance. As an example, the hypervisor interface can be as efficient as a system call, while hardware virtualization extensions require many cycles to trap, decode, and emulate each instruction.

> If you're curious, the Linux code to switch address spaces can be found in the `activate_mm` macro in `arch/x86/include/asm/context.h`. On regular hardware it calls `switch_mm` which executes a `LOAD CR3` instruction; in paravirtualized mode it calls `paravirt_activate_mm` (in `arch/x86/include/asm/paravirt.h`) which invokes a "hypercall" to request the hypervisor to perform the operation.

For years Amazon EC2 used a modified version of the Xen paravirtualized hypervisor, although as hardware virtualization support continues to improve, this remains the case only for a small number of their instance types.

What type of virtualization is fastest? This is actually a hard question—putting something (like virtualization support) into hardware doesn't automatically make it faster. State-of-the-art hardware and software-based (binary translation) hypervisors can have equivalent performance[7], so the choice between them often comes down to features.

Review questions

8.5.1. Which of the following are correct?

a) Paravirtualization requires specialized hardware support
b) Paravirtualization provides a system call-like interface that the guest OS uses to e.g. switch page tables
c) Paravirtualization requires modification to the guest operating system

[7] citation here - Ageson

8.6 Paravirtualized I/O Devices

It is common for hypervisors to have optional drivers (VMware Tools, VirtualBox Guest Additions, etc.) which can be loaded in the guest to improve performance. These typically include paravirtualized drivers for the disk controller and network interface: rather than catching writes to emulated registers, the paravirtualized driver uses a system call-like interface to make I/O requests to the hypervisor. Note that this works because almost all operating systems provide a simple means to load arbitrary kernel-mode drivers for third-party hardware; a paravirtualized device is just another piece of (virtualized) hardware that you need to install a driver for. In contrast, operating systems writers don't typically anticipate the need to support plugging in a different type of CPU.

8.7 Linux Containers and Docker

Running different applications within separate virtual machines provides a number of benefits when compared to running them all on the same unvirtualized operating system:

- Security—if one application is compromised, or is untrusted, the only way for it to attack the other applications (other than via the external network) is by subverting the hypervisor. This is difficult, as they are small and have tended to be quite reliable in practice. (i.e. with few bugs that can be exploited)
- Performance isolation—server-class virtualization systems can enforce resource limits (memory, CPU time, disk and network I/O) which ensure that heavy loads on one application do not negatively impact another.
- Management isolation—each virtual machine has its own file system, administrative (root) account, installed libraries, etc. and can be configured without regard to the dependencies of other applications running in other virtual machines.
- Packaging convenience—a virtual machine image is a convenient and useful format for storing a virtual machine and all of its configuration, as well as for distributing it to others. (like the CS-5600 virtual machine image you received at the beginning of this class)

Note, however, that none of these benefits actually *requires* hardware virtualization[8]. If all of the applications are going to be running on the same operating system, then *Operating System Virtualization* can be used: rather than pretending that a single hardware machine is actually multiple

[8]That is, unless you need to run multiple different operating systems.

virtual machines, we pretend that a single instance of the operating system is actually multiple instances. This approach was first used in FreeBSD *jails* and Solaris *containers*, but the mostly widely known example today is Linux Containers (LXC) and Docker.

LXC allows the creation of isolated process groups: each process in such a group (and any children of those processes) thinks that the group has the entire operating system to itself. This is done via two mechanisms:

Namespaces - in recent Linux versions, any access to the file system, process ID, networking, user or group IDs, or several more obscure system parameters (e.g. hostname) is relative to a *namespace*. In a normal system with no containers there will be a single namespace, visible to all processes. (or at least those that have sufficient permission, in the case of e.g. file system access) However you can also create new namespaces, with a restricted view of the file system (e.g. only able to see a small subtree), with their own process ID space and user names and IDs, and separate network interfaces and addresses. Within a namespace you can have a root user which can perform privileged operations within the namespace, but which has no power or visibility outside of it.

Control groups (cgroups) - these are used to control operating system allocation of resources such as memory, CPU time, or disk and network bandwidth. A cgroup can be associated with a process group, and the process group as a whole will be subject to any limits (e.g. on memory, CPU time, etc.) placed on that cgroup.

The combination of these two features allows the creation of separate *containers*, each with its own file system, network interfaces, etc., and where processes within a container are isolated from those in other containers or in the "base" or root operating system within which these containers were created. Processes in a container interact with the OS kernel in exactly the same way as in a non-containerized system; the only difference is in what they *see*, which is controlled via namespaces, and how their CPU time and I/O are *scheduled*, which is controlled via cgroups. Containers are thus more efficient, as there is no virtualization overhead, and can be created almost as quickly as normal processes.

Since there is still a single operating system kernel, all containers in a system share the same operating system version. Note, however, that they may have entirely different file systems; thus it is quite possible to have both a Red Hat and an Ubuntu distribution running in separate containers on the same machine, although each will be using the kernel of the underlying system.

Docker is based on LXC; however perhaps its main innovation is the way in

which it manages container file systems. A Docker container uses a *union file system* to join together multiple file systems—the first one of which is writeable, and with one or more read-only ones "behind" it. To understand the operation of a union file system, consider how the Unix shell finds an executable: it searches each directory in the PATH variable in order, and takes the first version of the file that it finds. Thus if the value of PATH is `/usr/bin:/sbin:/bin`, and you type `ls`, it will search `/usr/bin/ls` (not found), `/sbin/ls` (not found), and then `/bin/ls` (successful). A union file system operates in a similar fashion: on read access to a file (or directory) it will search through each underlying file system in order until it is found. When writing to a file, however, it will write to the first writable file system in the list, providing a form of copy-on-write.

This allows various environments to be *stacked*: e.g. a base file system containing the files from a minimal Linux installation, with additional file systems "on top" of it containing installed versions of various packages one wishes to use, and a writable file system on top for per-instance configuration parameters, application data, etc.

Answers to Review Questions

8.0.1 (4), all of the above.

8.1.1 (1) False: that's why it's called software emulation. (2) True: in fact it's much slower.

8.1.2 (1). privileged instructions will trap in user mode, and the hypervisor emulates them.

8.2.1 (3). The guest cannot be allowed to manipulate hardware page tables, and the hypervisor does not know the guest mappings, but the hypervisor can compose the "fake" guest page tables with hypervisor mappings to provide the correct translation.

8.5.1 (2) and (3). The hardware interface is replaced with "hypercalls", and the guest OS must be modified to use them instead of direct hardware access.