What do VMMs enable?

• Running multiple operating systems (called “guest OS’s”) and their applications on a single physical computer, as if each were running on its own private virtual computer

• Efficient – mostly direct execution, rather than simulation

• Contemporary examples
  – VMware
  – Microsoft’s VirtualPC / VirtualServer
  – Parallels (Mac)
  – Xen
VMM structure

Virtual Machine = Guest OS + apps

Virtual Machine = Guest OS + apps

applications

Windows

virtual machine monitor

hardware

applications

Linux
Basic ideas

• Guest OS runs in user mode
• When any kind of interrupt / exception / trap occurs, we’ll end up in the VMM rather than the guest OS
• VMM simulates state changes that would have been made by the hardware, then restarts VM at the guest OS handler address
  – E.g., stuffs the saved PC where the architecture says it should be
• When the guest OS tries to execute a privileged instruction
  – VMM gets control, simulates effect of privileged instruction
    • VMM knows that guest OS was in virtual kernel mode so the attempted operation is OK
VMM History

• Conceived by IBM in the late 1960’s
  – CP-40, CP-67, VM/360
• Sold continuously since then
• Used first for OS development and debugging, then for time sharing (multiple single-user OS’s, plus a few single-job batch OS’s), eventually for server consolidation
VMMs Today

• OS development and debugging
• Software compatibility testing
• Running software from another OS
  – Or, OS version
• Virtual infrastructure for Internet services (server consolidation)

• Examples
  – Run Windows on your Mac, or MacOS on your PC
  – VMware in CSE 451
  – Amazon’s Elastic Compute Cloud (EC2)
Comparing the Unix and VMM APIs

<table>
<thead>
<tr>
<th></th>
<th>UNIX</th>
<th>VMM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage</strong></td>
<td>File system</td>
<td>(virtual) disk</td>
</tr>
<tr>
<td><strong>Networking</strong></td>
<td>Sockets</td>
<td>(virtual) Ethernet</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>Virtual Memory</td>
<td>(virtual) Physical memory</td>
</tr>
<tr>
<td><strong>Display</strong></td>
<td>/dev/console</td>
<td>(virtual) Keyboard, display device</td>
</tr>
</tbody>
</table>
Possible Implementation Strategy: Complete machine emulation

- The VMM implements the complete hardware architecture in software

```java
while(true) {
    Instruction instr = fetch();
    // emulate behavior in software
    instr.emulate();
}
```

Drawback: This is really slow
Practical alternative: VMM gets control on privileged instructions only

- Treat guest operating systems (and their apps) like an application
  - Guest OS (and its apps) run in user mode
  - Most instructions execute natively on the CPU
  - Privileged instructions are trapped and emulated
Virtualizing the User/Kernel Boundary

• Both the guest OS and applications run in (physical) user-mode
• For each virtual machine, the VMM keeps a software mode bit:
  – During a system call, switch to “kernel” mode
  – On system call return, switch to “user” mode
• What does the VMM do if a VM executes a privileged instruction while in virtual user mode?
• What does the VMM do if a VM executes a privileged instruction while in virtual kernel mode?
Tracing Through a File System Read

Application

Guest OS

VMM

Hardware

read() syscall

trap detected

trap handler;
change VM to “kernel” mode

trap handler;
handle read syscall
read from disk()

priv insc. detected

trap handler;
emulate I/O

...
Questions, to clarify …

• What if the I/O could be handled from the buffer cache?
• Does the VMM handle a VM’s I/O request synchronously?
• There are a zillion different types of disks (and networks and …) … Do the device drivers for these reside in the guest OS or in the VMM?
A possible “gotcha”

- All instructions that modify hardware state must be privileged (so that VMM can get control, modify the virtual hardware state for that guest, and not modify the physical hardware state)
- Example: Suppose the ERET instruction (return to a user process after handling an exception) is not privileged
  - ERET sets the PC to the saved PC, and sets CPU mode to user
  - There doesn’t seem to be a reason to prevent user processes from doing this (even if there’s no reason for them to want to)

Why would this be a problem for a VMM?
x86

- Conditions for an architecture to be virtualizable were defined in 1974
- x86 architecture did not satisfy these conditions!
  - Many reasons, but most of them stem from instructions that have different behavior in user mode and kernel mode, and that don’t trap when executed in user mode

- Approach: binary re-writing
  - When a code page is loaded, scan it, looking for offending instructions
  - Patch these to cause a fault
  - Remember the instruction that used to be there
Other approaches

- **Hardware**: Both Intel (VT-x) and AMD (AMD-V) have developed virtualization extensions to the architecture (starting ~2006)
- **Paravirtualization**: Export a slight modification of the hardware; port the OS to this new hardware
Memory

- VMM’s also utilize memory protection (in addition to privileged instructions) to do their job
- Have not described how memory is virtualized by a VMM, creating “virtual physical memory” for the guest OS’s
- Approach involves the VMM futzing with the page tables in the guest OS’s
Trust Issues

Problem:

– Who can you trust?
– OS protects processes from each other
  • OS is “trusted” since you’re running it on your hardware
  • You don’t worry about OS snooping your data
– But in the cloud, Amazon (or Microsoft or Google) are running your operating systems in their VMM
  • VMs are ”just” user mode processes
  • VMM “naturally” isolates them
  • But, the VMM can look into the guest OS/process!
How You Can Trust The VMM

Solution:

- Tricky hardware!
- Keep all data encrypted
  - On disk, no problem.
  - In memory, sure… but...
  - How does the processor read/write/execute?
- Intel SGX/MEE processor / memory controller
  - RAM is encrypted!
  - Special instructions to tell processor where the encrypted regions are
  - Processor decrypts pages into hidden caches and executes from there