VMM – a Brief History

• Virtual Machine Monitor: a software-abstraction layer that partitions the HW into one or more virtual machines

• 1960s: used for multiplexing the scarce general purpose computing platforms among multiple applications

• 1980s: multitasking OSes + low HW costs
  – Rendered VMMs obsolete
  – Consequently, no hardware support for virtualization in the CPU architectures of the time (e.g., x86)
And now…

Compared to “cloud computing” (in red)
Why this revival?

• Virtual Machine Monitor: a software-abstraction layer that partition the HW into one or more virtual machines
• 1960s: used for multiplexing the scarce general purpose computing platforms among multiple applications
• 1980s: multitasking OSes + low cost hardware
  – Rendered VMMs obsolete
• 2000s: multitasking OSes + low cost hardware
  – Revived VMMs
Cause and Solution

• Increased OS functionality:
  – More capable OSes:
    • Fragile and vulnerable

• Low cost hardware:
  – Proliferation of machines:
    • Underused
    • With high space and management overhead

• Solution: back to one application per machine
  – Per virtual machine
  – This time: VMM as a means of multiplexing hardware for server consolidation
    • Solution for security and reliability
VMM: Decoupling the HW and the OS by a layer of indirection

- Uniform view of underlying HW,
  - so virtual machines can run on any hardware
- Complete encapsulation of a virtual machine’s software state,
  - so migration is much easier
- Total mediation of all interactions between the virtual machine and the underlying HW,
  - thus allowing strong isolation between VMs
Big Picture (and Terminology)

Virtual Machine

Guest OS

Operating system

Operating system

Operating system

Virtual machine monitor

Hardware
Xen

(Original slides by Kishore Ramachandran adapted by Anda Iamnitchi)
Key points

• **Goal**: extensibility and flexibility akin with Exokernel/Micro-kernel goals

• **Main difference**: granularity of operating systems rather than applications
  – running several commodity operating systems on the *same* hardware simultaneously without sacrificing performance or functionality

• **Why?**
  – Application mobility
  – Server consolidation
  – Co-located hosting facilities
  – Distributed web services
  – Secure computing platforms
Possible Virtualization Approaches

- Standard OS (such as Linux, Windows)
  - Meta services (such as grid) for users to install files and run processes
  - Administration, accountability, and performance isolation become hard

- Retrofit performance isolation into OSs
  - Accounting resource usage correctly can be an issue unless done at the lowest level and very fine granularity (e.g., Exokernel)

- Xen approach
  - Multiplex physical resource at OS granularity
Full Virtualization

- Make hw completely invisible to the OS
- Virtual hardware identical to real one
  - Relies on hosted OS trapping to the VMM for privileged instructions
  - Pros: run unmodified OS binary on top
  - Cons:
    - supervisor instructions can fail silently in some hardware platforms (e.g. x86)
      - Solution in VMware: Dynamically rewrite portions of the hosted OS to insert traps
    - need for hosted OS to see real resources: real time, page coloring tricks for optimizing performance, etc…
Paravirtualization

• Presents a virtual machine abstraction similar *but not identical* to the underlying hardware
  – Pros:
    • allows strong resource isolation on uncooperative hardware (x86)
    • enables optimizing guest OS performance and correctness
  – Cons:
    • Need to modify the guest OS
(More) Background

• ABI, signals, interrupts, system calls
  – ABI defines the machine as seen by process

• The non-virtualizable x86 architecture
  – Visibility of privileged state: the guest can observe it has been deprivileged when reads its code segment selector
  – Lack of traps when privileged instructions run at unprivileged level: instructions fail silently
Xen Principles

• Support for unmodified application binaries
• Support for multi-application OS
  – Complex server configuration within a single OS instance
• Use paravirtualization in order to support uncooperative architectures such as x86
• Enable guest OSes to optimize performance and correctness
Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including Domain0 running control software in a XenoLinux environment.
Challenges

• CPU virtualization
  – ...
• Memory virtualization
  – ...
• I/O virtualization
  – ...

Xen: Memory Management

• What would make memory virtualization easy
  – Software TLB
  – Tagged TLB =>no TLB flush on context switch
    • Tag identifies a different address space
X86 does not have either

• Xen approach
  – Guest OS responsible for allocating and managing hardware
    page tables
  – Every guest OS has its own address space
  – Xen occupies top 64MB of every address space. Why?
    • To save moving between address spaces (hypervisor calls), no TLB
      flush
    • Xen code and its data structures
    • Cost?
Physical Memory

• At domain creation, hardware pages “reserved”
• Domain can increase/decrease its quota
• Xen does not guarantee that the hardware pages are contiguous
• Guest OS can maintain its own mapping of “contiguous” “physical memory” mapped to the “hardware pages”
• Creating a new Page Table by Guest OS
  – (every process has its own page table)
  – allocate and initialize a page and registers it with Xen to serve as the new page table
  – all subsequent updates to this page via Xen
  – can batch updates to reduce the cost of entering and exiting Xen
• Segmentation by guest OS virtualized similarly
Virtual Address Translation

• Xen solution
  – Physical memory is not contiguous for guest OS
    • Fundamental shift from the traditional view of the HW
  – Register guest OS PT directly with MMU
  – Guest OS has read only access to the PT
    • All modifications to the PT via Xen
    • Why?
  – Type associated with a page frame
    • PD, PT, LDT, GDT, RW (terminology specific to x86 architecture)
    • All except RW: read-only access to the page for guest OS
    • Guest OS can retask a page only when ref count is 0
Xen: CPU Virtualization

- Four privilege levels in x86
  - Ring 0: originally for OS (now Xen)
  - Ring 1: originally not used (now guest OS)
  - Ring 3: originally for applications (now applications of the guest OS)
  - Ring 2: originally not used (now for shared libraries for supporting user processes in Ring 3)

- Privileged instructions
  - Validated and executed in Xen (e.g., installing a new PT)

- Exceptions
  - Registered with Xen once. Accepted (validated) if don’t require to execute exception handlers in ring 0.
  - Called directly without Xen intervention
  - All syscalls from apps to guest OS handled this way (and executed in ring 1)

- Page fault handlers are special
  - Faulting address can be read only in ring 0
  - Xen reads the faulting address and passes it via stack to the OS handler in ring 1
CPU Scheduling

• Bounded Virtual Time (BVT) scheduling algorithm [Duda & Cheriton, SOSP 99]
  – BVT guarantees fairness over a window of time
  – Keeps track of CPU time allocated to each guest OS.
    • The guest OS uses its own scheduling techniques for its processes.
Time and Timers

• Guest OSs have access to
  – Real time (cycle counter accuracy for real time tasks in guest OSs)
  – Virtual time (enables time slicing within the guest OS)
  – Wall clock time (real time + domain changeable offset)
    • Time zones, particular settings (e.g., 5 minutes fast)

• Guest OS maintain their own internal timer queues and use a pair of Xen timers (real and virtual)
Xen: Device I/O Virtualization

• Set of clean and simple device abstractions
• Allows buffers to be passed directly to/from guest OS to I/O devices via shared memory with Xen
• I/O rings implement event delivery mechanism for sending asynchronous notifications to the guest OS
<table>
<thead>
<tr>
<th>OS subsection</th>
<th># lines</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linux</td>
<td>XP</td>
<td></td>
</tr>
<tr>
<td>Architecture-independent</td>
<td>78</td>
<td>1299</td>
<td></td>
</tr>
<tr>
<td>Virtual network driver</td>
<td>484</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Virtual block-device driver</td>
<td>1070</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Xen-specific (non-driver)</td>
<td>1363</td>
<td>3321</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2995</td>
<td>4620</td>
<td></td>
</tr>
<tr>
<td>(Portion of total x86 code base)</td>
<td>1.36%</td>
<td>0.04%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The simplicity of porting commodity OSes to Xen. The cost metric is the number of lines of reasonably commented and formatted code which are modified or added compared with the original x86 code base (excluding device drivers).
Details of subsystem virtualization

• Control transfer
  – Between the guest OS and Xen
  – E.g., page table updates and creation

• Data transfer
  – Passing data between Xen and the OS
  – E.g., the fault address

These are used in the virtualization of all the subsystems
Control Transfer

• Hypercalls from guest OS to Xen
  – E.g., (set of) page table updates
• Events for notification from Xen to guest OS
  – E.g., data arrival on network; virtual disk transfer complete
• Events may be deferred by a guest OS (similar to disabling interrupts)
• Hypercalls can be aggregated to save on control transfers between guest OS and Xen
Data transfer – I/O rings

- Resource management and accountability
  - CPU time
    - Demultiplex data to the domains quickly upon interrupt
    - Account computation time for managing buffers to the appropriate domain
  - Memory buffers
    - Relevant domains provide memory for I/O to Xen
      - Protection guaranteed between different guest OSs
    - Xen pins page frames during I/O transfer
      - Used to defer interruptions on data transfers
    - One I/O ring for each device for each guest OS: e.g., one for writing data to the disk; one for sending network packets; one for receiving network packets; etc.
Figure 2: The structure of asynchronous I/O rings, which are used for data transfer between Xen and guest OSes.

I/O descriptors indirectly reference the actual memory buffers.
Each request: unique ID from guest OS
Response: use this unique ID
Network packets
– Represented by a set of requests
– Responses to these signal packet reception
Disk requests
– May be reordered for performance (e.g., elevator)
No copying between Xen and the guest OS (therefore, no performance penalties)
Domain may queue up multiple entries before invoking a hypercall request
Ditto for responses from Xen
I/O rings in memory shared by Xen and guest OS
Network

• Each guest OS has two I/O rings for network
  – One for receive and one for transmit
  – Each ring is a contiguous region of memory

• Transmit
  – Queue descriptor on the transmit ring
    • Points to the buffer in guest OS space
    • Page pinned till transmission complete
  – Round robin packet scheduler across domains

• Receive
  – Network packets written in the receive ring of the destination guest OS
  – Xen makes an upcall to the guest OS

• No copying!
Disk

• Batches of requests from competing domain taken and scheduled
• Since Xen has knowledge of disk layout, requests may be reordered
• No copying into Xen
• “Reoder barrier” to prevent reordering (may be necessary for higher level semantics such as write ahead log)
  – It will overwrite the order of the circular I/O memory buffer
<table>
<thead>
<tr>
<th>Memory Management</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space.</td>
</tr>
<tr>
<td>Paging</td>
<td>Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontiguous machine pages.</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td></td>
</tr>
<tr>
<td>Protection</td>
<td>Guest OS must run at a lower privilege level than Xen.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handlers remain the same.</td>
</tr>
<tr>
<td>System Calls</td>
<td>Guest OS may install a ‘fast’ handler for system calls, allowing direct calls from an application into its guest OS and avoiding indirecting through Xen on every call.</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Hardware interrupts are replaced with a lightweight event system.</td>
</tr>
<tr>
<td>Time</td>
<td>Each guest OS has a timer interface and is aware of both ‘real’ and ‘virtual’ time.</td>
</tr>
<tr>
<td><strong>Device I/O</strong></td>
<td></td>
</tr>
<tr>
<td>Network, Disk, etc.</td>
<td>Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications.</td>
</tr>
</tbody>
</table>

Table 1: The paravirtualized x86 interface.
Figure 3: Relative performance of native Linux (L), XenoLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).