Outline

• Last lecture: basics of virtualization
  • VMM is an OS that maintains a machine-like interface instead of a process interface
  • Many reasons to use virtualization
  • Originally, virtualization wasn’t thought possible on x86
  • VMware introduced binary translation

• This lecture: recent developments
  • More detailed discussion of HW support for virtualization
  • Safe user-level access to privileged CPU features
Intel VT-x

- Makes x86 hardware “classically virtualizable” (as defined by Popek and Goldberg)
- Goal: **Direct execution** of most privileged instructions
- Introduces two CPU modes:
  - VMX root mode: for running VMM
  - VMX non-root mode: for running VMs (guest)
  - Each mode has its own rings (CPL0-CPL3)
- In-memory structure called VM Control Structure (VMCS) stores privileged register state and control flags
Intel VT-x

Non-Root Mode

CPL 3
CPL 2
CPL 1
CPL 0

User program
Guest OS

#vmlaunch
#vmresume

#vmexit

Root Mode

VMCS
VM enter and VM exit

- Transitions between VMX root mode and VMX non-root mode
- VM Exit
  - `vmcall` instruction
  - EPT page faults
  - Some trap-and-emulate (configured in VMCS)
  - Interrupts
- VM Enter
  - `vmlaunch` instruction: Enter VMX non-root mode for a new VMCS
  - `vmresume` instruction: Enter for the last VMCS
- Typical `vmexit/vmenter` is ~200 cycles on modern HW
Intel Extended Page Tables (EPT)

• **Goal:** Direct execution of guest page-table interactions
  • Reads and writes to page table in memory
    - `mov %eax, %cr3`
    - `invlpg`, etc.
• **Idea:** maintain two layers of paging translation
  • Normal page-table: Guest-virtual to guest-physical
  • EPT: guest-physical to host-physical
SR-IOV + IOMMU

- **Goal:** Allow direct execution of I/O device access

- **Challenge #1:** how to partition a single device into multiple instances
  - SR-IOV allows a PCI device to expose multiple, separate memory-mapped I/O regions

- **Challenge #2:** How to prevent DMA from overwriting memory belonging to VMM or another guest
  - IOMMU: Provides paging translation across PCIe bus
Big picture

• **Direct execution reduces overhead**
  - Avoids VM exits, trap-and-emulate, binary translation

• **Enabled by three microarchitectural changes:**
  - Intel VT-x: direct execution of most privileged instructions (e.g., IDT, GDT, CPL, EFLAG, etc.)
  - Intel EPT: direct execution of page table manipulation
  - IOMMU + SRIOV: direct execution of I/O interactions (e.g., network)
Operating systems today

App → OS → Hardware

App → OS → Hardware

App → OS → Hardware
What if you could give process access to raw hardware?

Access to existing Linux abstractions

Access to full HW capabilities

Hardware

OS

App

App

App
Dune

• Key idea: Use VT-x, EPT, etc, to support Linux processes instead of virtual machines
• Dune is a loadable Linux kernel module, makes it possible for an ordinary process to switch to “Dune mode”
• Dune mode processes can run alongside ordinary processes. Within a process, some threads can be Dune mode even if others aren’t
A Dune process

- Is still a Linux process
  - Has memory
  - Can make system calls
  - Is fully isolated
  - ...

- But isolated with VT-X non-root mode
  - Rather than with CPL=3 and page table protections

- Memory protection via EPT
  - Dune configures EPT so processes can only access the same physical pages it would normally have access to.
Why isolate a process with VT-x?

- Process can access all of Linux environment while also executing most privileged instructions
- User code now runs at CPL 0
- Process manages its own page table: %CR3
- Fast exceptions (e.g., page faults) via shadow IDT
  - Kernel crossings eliminated
  - Some interrupts configured to dispatch to non-root mode
  - Others configured to cause vmexit
- Can run sandboxed code at CPL 3
  - So process can act like a kernel!
How to perform a Linux system call from a Dune process?

• **int $80** just traps inside process at handler specified in shadow IDT

• **vmcall** instruction forces a VM exit
  • Dune module vectors exit into kernel system call table

• **Challenge: compatibility**
  • Existing code and libraries don’t use **vmcall**

• **Solution:**
  • Shadow IDT handler forwards the system calls it catches using **vmcall**
How to perform a Linux system call from a Dune process?

```
vmcall
int $80
```
Microbenchmarks: overheads

- **Two sources of overhead:**
  - VM exits and VM enters
  - EPT translations

<table>
<thead>
<tr>
<th></th>
<th>getpid</th>
<th>page fault</th>
<th>page walk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>138</td>
<td>2,687</td>
<td>35.8</td>
</tr>
<tr>
<td>Dune</td>
<td>895</td>
<td>5,093</td>
<td>86.4</td>
</tr>
</tbody>
</table>

Table 2: Average time (in cycles) of operations that have overhead in Dune compared to Linux.
Microbenchmarks: speedups

- Large opportunities for optimization
  - Faster system-call interposition and traps
  - More efficient user-level virtual memory manipulation

<table>
<thead>
<tr>
<th></th>
<th>ptrace</th>
<th>trap</th>
<th>appel1</th>
<th>appel2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>27,317</td>
<td>2,821</td>
<td>701,413</td>
<td>684,909</td>
</tr>
<tr>
<td>Dune</td>
<td>1,091</td>
<td>587</td>
<td>94,496</td>
<td>94,854</td>
</tr>
</tbody>
</table>

Table 3: Average time (in cycles) of operations that are faster in Dune compared to Linux.
Example: sandboxed execution

- Suppose your browser wants to run a plugin
  - Could be buggy or malicious
- Need a way to execute plugin but limit:
  - System calls
  - Memory access
- Using Dune:
  - Browser is a Dune thread:
    - Run at CPL0
  - Create a Dune thread for plugin:
    - PTE_U mappings only for allowed access
    - Run at CPL3
    - Can run system calls but they trap to browser
      - Browser filters or emulates system calls
What if you could give process access to raw hardware?

Browser (Dune CPL 0)  
Plugin (CPL 3)  
Plugin (CPL 3)  
Linux Kernel  
Hardware  
vi
Sandbox: SPEC2000 performance

- Only notable end-to-end effect is EPT overhead
- Can be eliminated through the use of large pages
More thoughts on use cases

- **Dune provides similar benefits to Exokernel**
  - Raw access to paging hardware for Appel&Li paper
  - Speed improvements alone may make some ideas more feasible (e.g., GC)

- **Each Dune thread can have a different page table!**
  - E.g., stthreads: a mechanism for least privilege
Summary

• VT-x, EPT, and SR-IOV/IOMMU enable direct execution of (most) guest instructions
• Dune implements processes with VT-x and EPT rather than ordinary ring protection
• Dune processes can use both Linux system calls and privileged HW
  • Enables fast access to page table and page faults
  • Enables processes to build kernel-like functionality
    - E.g., sandboxing untrusted plugins in CPL3
    - Hard to do this at all in Linux, let alone efficiently
Question

• How can a Dune process be in ring 0, but non-VMX root?
• Why would a child process not want to be in Dune mode?
• Why would we have some threads be in Dune mode, and others not?
• Is Dune emulating the privileged calls to, e.g., the TLB, or is it really changing the values in the hardware?
• What is `ioctl` (used by a process to enter Dune mode)?
• If Dune module is like VMM and Dune process like guest OS, how can Dune module “override” the normal host VMM