Traditional operating system
Virtual machines
A bit of history

- Virtual machines were popular in 60s-70s
  - Share resources of mainframe computers [Goldberg 1974]
  - Run multiple single-user operating systems
- Interest is lost by 80s-90s
  - Development of multi-user OS
  - Rapid drop in hardware cost
- Hardware support for virtualization was lost
What is the problem?

- Hardware is not designed to be multiplexed
- Loss of isolation
Virtual machine

Efficient duplicate of a real machine
- Compatibility
- Performance
- Isolation
Trap and emulate
What needs to be emulated?

- CPU and memory
  - Register state
  - Memory state
- Memory management unit
  - Page tables, segments
- Platform
  - Interrupt controller, timer, buses
- BIOS
- Peripheral devices
  - Disk, network interface, serial line
x86 is not virtualizable

- Some instructions (*sensitive*) read or update the state of virtual machine and don't trap (*non-privileged*)
  - 17 sensitive, non-privileged instructions [Robin et al 2000]
## x86 is not virtualizable (II)

<table>
<thead>
<tr>
<th>Group</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access to interrupt flag</td>
<td><code>pushf, popf, iret</code></td>
</tr>
<tr>
<td>Visibility into segment descriptors</td>
<td><code>lar, verr, verw, lsl</code></td>
</tr>
<tr>
<td>Segment manipulation instructions</td>
<td><code>pop &lt;seg&gt;, push &lt;seg&gt;, mov &lt;seg&gt;</code></td>
</tr>
<tr>
<td>Read-only access to privileged state</td>
<td><code>sgdt, sldt, sidt, smsw</code></td>
</tr>
<tr>
<td>Interrupt and gate instructions</td>
<td><code>fcall, longjump, retfar, str, int &lt;n&gt;</code></td>
</tr>
</tbody>
</table>

### Examples

- **`popf` doesn't update interrupt flag (IF)**
  - Impossible to detect when guest disables interrupts
- **`push %cs` can read code segment selector (%cs) and learn its CPL**
  - Guest gets confused
Solution space

- Parse the instruction stream and detect all sensitive instructions dynamically
  - Interpretation (BOCHS, JSLinux)
  - Binary translation (VMWare, QEMU)
- Change the operating system
  - Paravirtualization (Xen, L4, Denali, Hyper-V)
- Make all sensitive instructions privileged!
  - Hardware supported virtualization (Xen, KVM, VMWare)
    - Intel VT-x, AMD SVM
Basic blocks of a virtual machine monitor: QEMU example
Interpreted execution: BOCHS, JSLinux
What does it mean to run guest?

- Bochs internal emulation loop
- Similar to non-pipelined CPU like 8086

- How many cycles per instruction?
Binary translation: VMWare/QEMU
```c
int isPrime(int a) {
    for (int i = 2; i < a; i++) {
        if (a % i == 0) return 0;
    }
    return 1;
}
```
isPrime:  mov  %ecx, %edi ; %ecx = %edi (a)
    mov  %esi, $2 ; i = 2
    cmp  %esi, %ecx ; is i >= a?
    jge  prime ; jump if yes
nexti:  mov  %eax, %ecx ; set %eax = a
    cdq ; sign-extend
    idiv  %esi ; a % i
    test  %edx, %edx ; is remainder zero?
    jz   notPrime ; jump if yes
    inc  %esi ; i++
    cmp  %esi, %ecx ; is i >= a?
    jl   nexti ; jump if no
prime:  mov  %eax, $1 ; return value in %eax
    ret
notPrime: xor  %eax, %eax ; %eax = 0
    ret

isPrime’:  mov  %ecx, %edi ; IDENT
             mov  %esi, $2
             cmp  %esi, %ecx
             jge  [takenAddr] ; JCC
             jmp  [fallthrAddr]
isPrime':
  *mov   %ecx, %edi   ; IDENT
  mov   %esi, $2
  cmp   %esi, %ecx
  jge   [takenAddr]   ; JCC
  ; fall-thru into next CCF
nexti':
  *mov   %eax, %ecx   ; IDENT
  cdq
  idiv  %esi
  test  %edx, %edx
  jz    notPrime'    ; JCC
  ; fall-thru into next CCF
  *inc   %esi   ; IDENT
  cmp   %esi, %ecx
  jl     nexti'      ; JCC
  jmp   [fallthrAddr3]

notPrime':
  *xor   %eax, %eax   ; IDENT
  pop    %r11        ; RET
  mov    %gs:0x0835eb8(%rip), %rcx   ; spill %rcx
  movz x  %ecx, %r11b
  jmp    %gs:0xfc7d5e0(8*%rcx)
Interpreted execution revisited: Bochs
Instruction trace cache

- How to make this loop faster?
Instruction trace cache

- 50% of time in the main loop
- Fetch, decode, dispatch
- Trace cache (Bochs v2.3.6)
  - Hardware idea (Pentium 4)
  - Trace of up to 16 instructions (32K entries)
- 20% speedup
Improve branch prediction

- 20 cycles penalty on Core 2 Duo

```c
void BX_CPU_C::SUB_EdGd(bxInstruction_c *i) {
    Bit32u op2_32, op1_32, diff_32;

    op2_32 = BX_READ_32BIT_REG(i->nnn());

    if (i->modC0()) {
        // reg/reg format
        op1_32 = BX_READ_32BIT_REG(i->rm());
        diff_32 = op1_32 - op2_32;
        BX_WRITE_32BIT_REGZ(i->rm(), diff_32);
    } else {
        // mem/reg format
        read_RMW_virtual_dword(i->seg(),
                                RMAddr(i), &op1_32);
        diff_32 = op1_32 - op2_32;
        Write_RMW_virtual_dword(diff_32);
    }

    SET.LAZY_FLAGS_SUB32(op1_32, op2_32, diff_32);
}
```
Improve branch prediction

- Split handlers to avoid conditional logic
  - Decide the handler at decode time (15% speedup)
Resolve memory references without misprediction

- Bochs v2.3.5 has 30 possible branch targets for the effective address computation
  - Effective Addr = (Base + Index*Scale + Displacement) mod(2^AddrSize)
  - e.g. Effective Addr = Base, Effective Addr = Displacement
  - 100% chance of misprediction

- Two techniques to improve prediction:
  - Reduce the number of targets: leave only 2 forms
  - Replicate indirect branch point

- 40% speedup
## Time to boot Windows

<table>
<thead>
<tr>
<th></th>
<th>1000 MHz Pentium III</th>
<th>2533 MHz Pentium 4</th>
<th>2666 MHz Core 2 Duo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bochs 2.3.5</td>
<td>882</td>
<td>595</td>
<td>180</td>
</tr>
<tr>
<td>Bochs 2.3.6</td>
<td>609</td>
<td>533</td>
<td>157</td>
</tr>
<tr>
<td>Bochs 2.3.7</td>
<td>457</td>
<td>236</td>
<td>81</td>
</tr>
</tbody>
</table>
## Cycle costs

<table>
<thead>
<tr>
<th></th>
<th>Bochs 2.3.5</th>
<th>Bochs 2.3.7</th>
<th>QEMU 0.9.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register move (MOV, MOVSX)</td>
<td>43</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>Register arithmetic (ADD, SBB)</td>
<td>64</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>Floating point multiply</td>
<td>1054</td>
<td>351</td>
<td>27</td>
</tr>
<tr>
<td>Memory store of constant</td>
<td>99</td>
<td>59</td>
<td>5</td>
</tr>
<tr>
<td>Pairs of memory load and store operations</td>
<td>193</td>
<td>98</td>
<td>14</td>
</tr>
<tr>
<td>Non-atomic read-modify-write</td>
<td>112</td>
<td>75</td>
<td>10</td>
</tr>
<tr>
<td>Indirect call through guest EAX register</td>
<td>190</td>
<td>109</td>
<td>197</td>
</tr>
<tr>
<td>VirtualProtect system call</td>
<td>126952</td>
<td>63476</td>
<td>22593</td>
</tr>
<tr>
<td>Page fault and handler</td>
<td>888666</td>
<td>380857</td>
<td>156823</td>
</tr>
<tr>
<td>Best case peak guest execution rate in MIPS</td>
<td>62</td>
<td>177</td>
<td>444</td>
</tr>
</tbody>
</table>
Paravirtualization:
Xen
Full virtualization

- Complete illusion of physical hardware
  - Trap _all_ sensitive instructions
  - Example: page table update
Full virtualization

• Complete illusion of physical hardware
  • Trap _all_ sensitive instructions
  • Example: page table update
Full virtualization

- Complete illusion of physical hardware
  - Trap _all_ sensitive instructions
  - Example: page table update

```
if (safe) {
    update_pte();
    emulate_mov();
}
```
Performance problems

- Traps are slow
- Binary translation is faster
  - For some events
Paravirtualization

- No illusion of hardware
- Instead: paravirtualized interface
  - Explicit hypervisor calls to update sensitive state
    - Page tables, interrupt flag

- But Guest OS needs porting
  - Applications run natively in Ring 3
Paravirtualization

Paravirtualized OS

PTE update

Batch updates
update 1
update 2

Invoke hypervisor

Hypervisor

if (safe)
update
Hardware support for virtualization:
KVM
Basic idea

- Host instruction stream
- Guest instruction stream
- VM Entry
- VM Exit
- VMCS
- Guest State
- Host State
New mode of operation: VMX root

- VMX root operation
  - 4 privilege levels
- VMX non-root operation
  - 4 privilege levels as well, but unable to invoke VMX root instructions
  - Guest runs until it performs exception causing it to exit
  - Rich set of exit events
  - Guest state and exit reason are stored in VMCS
Virtual machine control structure (VMCS)

- Guest State
  - Loaded on entries
  - Saved on exits
- Host State
  - Saved on entries
  - Loaded on exits
- Control fields
  - Execution control, exits control, entries control
Guest state

- Register state
- Non-register state
  - Activity state:
    - active
    - inactive (HLT, Shutdown, wait for Startup IPI interprocessor interrupt))
  - Interruptibility state
Host state

• Only register state
  • ALU registers,
• also:
  • Base page table address (CR3)
  • Segment selectors
  • Global descriptors table
  • Interrupt descriptors table
VM-execution controls
(asynchronous events control)

External interrupts (maskable or IRQs) cause exits (yes/no)
If not, then they delivered through guest IDT

NMI cause exits (yes/no)
If not, then they are delivered normally through guest IDT (descriptor 2)
VM-execution controls
(synchronous events control, not all reasons are shown)
Exception bitmap
(one for each of 32 IA-32 exceptions)

- IA-32 defines 32 exception vectors (interrupts 0-31)
- Each of them is configured to cause or not VM-exit

Bit 31

Bit 0

14 – page fault
Nested page tables

- gCR3: paged by gCR3
- hCR3: paged by hCR3
- gPT: paged by gCR3
- hPT: paged by hCR3
- VMM
- Host Virtual
- Guest Virtual
- Host Physical
- CR3 used by VMM
- Translation can be cached in TLB
- CR3 used by VMM

- hPT
- gPT
- PT
Page table lookup

- 4-level page table
Efficient I/O
Where is the bottleneck

- What is the bottleneck in case of virtualization?
  - CPU?
    - CPU bound workloads execute natively on the real CPU
    - Sometimes JIT compilation (binary translation makes them even faster [Dynamo])
  - Everything what is inside VM is fast!

- What is the most frequent operation disturbing execution of VM?
  - **Device I/O**!
    - Disk, Network, Graphics
Virtual devices in Xen
Virtual devices in Xen

![Diagram showing Xen and dom0 (Linux) relationship]

- dom0 (Linux)
- Xen
Virtual devices in Xen
Virtual devices in Xen
Virtual devices in Xen
How to make the I/O fast?

- Take into account specifics of the device-driver communication
  - **Bulk**
    - Large packets (512B – 4K)
  - **Session oriented**
    - Connection is established once (during boot)
    - No short IPCs, like function calls
    - Costs of establishing an IPC channel are irrelevant
  - **Throughput oriented**
    - Devices have high delays anyway
  - **Asynchronous**
    - Again, no function calls, devices are already asynchronous
Shared rings and events

- **dom0 (Linux)**
  - Linux Block Driver
  - Backend
- **Xen**
  - Interrupt-like event channel
- **domU (Linux)**
  - FrontEnd

Shared page with a ring buffer
Shared rings

Receiver:
- rsp_prod_pvt
- req_cons
- nr_ents = 256
  *shared

Shared:
- req_prod
- rsp_prod

Sender:
- req_prod_pvt
- rsp_cons
- nr_ents = 256
  *shared
Shared rings

**Receiver:**
- `rsp_prod_pvt`
- `req_cons`
- `nr_ents = 256`
- *shared*

**Sender:**
- `req_prod_pvt`
- `rsp_cons`
- `nr_ents = 256`
- *shared*

![Diagram showing shared rings with receiver and sender processes, containing unconsumed requests and responses.](image-url)
Shared rings

Receiver:
- rsp_prod_pvt
- req_cons
- nr_ents = 256
*shared

Add requests:
- req_prod <-- req_prod_pvt

Sender:
- req_prod_pvt
- rsp_cons
- nr_ents = 256
*shared

Shared:
- req_prod
- rsp_prod

Unconsumed requests

Unconsumed responses
Shared rings

Check requests:
req_cons != req_prod

Receiver:
rsp_prod_pvt
req_cons
nr_ents = 256
*shared

Shared:
req_prod
rsp_prod

Add requests:
req_prod<--req_prod_pvt

Sender:
req_prod_pvt
rsp_cons
nr_ents = 256
*shared

Unconsumed requests

Unconsumed responses
Where is a performance bottleneck here?

Check requests:
req_cons \neq req_prod

Receiver:
rsp_prod_pvt
req_cons
nr_ents = 256
*shared

Shared:
req_prod
rsp_prod

Add requests:
req_prod\leftarrow req_prod_pvt

Sender:
req_prod_pvt
rsp_cons
nr_ents = 256
*shared

Unconsumed requests

Unconsumed responses
Eliminate cache thrashing

Check requests:
- req_cons != req_prod
- req_cons + 1 != NIL

Receiver:
- rsp_prod_pvt
- req_cons
- nr_ents = 256
  *shared

Shared:
- req_prod
- rsp_prod

Add requests:
- req_prod <-- req_prod_pvt
- req_prod_pvt + 1 = NIL

Sender:
- req_prod_pvt
- rsp_cons
- nr_ents = 256
  *shared
GPUs

• Sending frames from the framebuffer
  • No hardware acceleration
  • Too slow

• OpenGL/DirectX level virtualization
  • Send high-level OpenGL commands over rings
  • OpenGL operations will be executed on the real GPU
Devices supporting virtualization

Diagram showing the structure of virtualization with Xen, dom0 (Linux), and guests (Linux) with resources like Disk and Net.
References

• A Comparison of Software and Hardware Techniques for x86 Virtualization. Keith Adams, Ole Agesen, ASPLOS'06

• Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, Edward Y. Wang, ACM TCS'12.

• Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. Darek Mihocka, Stanislav Shwartsman, ISCA-35.