Windows Kernel Internals II
Virtual Machine Architecture

University of Tokyo – July 2004

Dave Probert, Ph.D.
Advanced Operating Systems Group
Windows Core Operating Systems Division
Microsoft Corporation

© Microsoft Corporation 2004
Hosted VM Model

Windows acts as a “host”
- Resources for each VM are allocated from the host
- All I/O with external devices is performed through the host

“Guest” code runs within a separate context
- Independent address space
- Specialized “VMM” kernel
VM Components

VMM Kernel

- Thin layer, all in assembly
- Code executed at ring-0
- Exception handling
- External Interrupt pass-through
- Page table maintenance
- Located within a 32MB area of address space known as the “VMM work area”
- Work area is relocatable
- One VMM instance per virtual processor
VM Components

**VMM Driver**
- Provides kernel-level VM-related services
  - CreateVirtualMachine
  - CreateVirtualProcessor
  - ExecuteVirtualProcessor
- Implements context switching mechanism between the host and guest contexts
- Loads and bootstraps the VMM kernel
- Much of the security work we’ve done recently involved repackaging the VMM kernel code into the VMM driver
VM Components

**NDIS Filter Driver**

- Allows VM to send and receive Ethernet packets via physical Ethernet hardware
- Spoofs unique MAC addresses for virtual NICs
- Injects packets into host Ethernet stack for guest-to-host networking

© Microsoft Corporation 2004
VM Components

Virtual PC / Virtual Server executables
- Device emulation modules
- Resource allocation
- VM configuration creation & editing
- VM control (start, stop, pause, save)
- Scripting APIs
- User interaction
- Host side of guest/host integration features
VM Components

Virtual Machine “Additions”

- Collection of components installed within the guest environment by the user
- Implement optimizations
  - Video
  - SCSI
  - Networking (in the future)
  - Guest kernel patches
- Implement guest half of guest/host integration features
  - Clipboard sharing
  - File drag and drop
  - Arbitrary video resizing
VM Execution Loop

Host code repeatedly calls ExecuteVirtualProcessor
VMM acts as “co-routine” (i.e. VMM state is saved and restored each time ExecuteVirtualProcessor is called)
Cycles spent inside guest context are counted against the calling thread
  – Host code can control how much time is spent in guest
Return code indicates why ExecuteVirtualProcessor returned
  – Time slice complete
  – IN or OUT instruction encountered
  – HLT instruction encountered
Processor Virtualization

x86 Virtualization

– Processor is non-virtualizable
  • Poor privileged and user state separation
    – For example, EFLAGS register contains condition codes (user state) and interrupt mask (privileged state)
  • Some instructions that access privileged state are non-trapping
– Overly complex and messy architecture
  • Many modes, legacy protection mechanisms and general “warts”
Processor Emulation

In general, emulation is necessary

– VM uses a binary translation mechanism
  • Most instructions are copied directly
  • Non-virtualizable ("dangerous") instructions are modified

– Binary translation execution imposes ~50% performance overhead
Direct Execution

In some processor modes, it’s safe to use direct execution, others require emulation

<table>
<thead>
<tr>
<th>Mode</th>
<th>Execution Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Mode</td>
<td>Emulation</td>
</tr>
<tr>
<td>Virtual 8086 (v86) mode</td>
<td>Direct Execution</td>
</tr>
<tr>
<td>Protected Mode Ring 3</td>
<td>Direct Execution (with a few exceptions)</td>
</tr>
<tr>
<td>Protected Mode Ring 0</td>
<td>Emulation, unless known to be safe</td>
</tr>
</tbody>
</table>
Direct Execution

“Ring Compression”
- Guest ring-0, 1, 2 code is executed at ring 1
- Guest ring-3 code is executed at ring 3
- Provides correct MMU protection semantics (since ring 0-2 can access privileged pages)

Direct execution of ring-0 code is only allowed if the VMM is notified that it’s “safe”
- This requires patching certain “dangerous” instruction sequences in the Windows kernel and HAL
- Patching is performed at runtime in memory only
- Patches are different for each version of Windows kernel & HAL
Guest OS Patching

Examples:
- PUSHFD / POPFD
- CLI / STI
- Spin lock acquisition failure (in the future)

Original Code

```
pushfd
cli
mov eax, [ebp+8]
call [eax]
popfd
ret
```

pushfd never traps (breaks IF virtualization)
cli traps, but cannot be easily patched with a jmp because it only takes up one byte
popfd never traps (breaks IF virtualization)

This sequence prevents correct behavior in direct execution
Guest OS Patching

Synthetic instructions

– Use an illegal instruction form (reserved for us by Intel)
– Five bytes in length (for ease in patching)
– Exhibit same side effects of real instruction

Original Code

```assembly
pushfd
cli
mov    eax, [ebp+8]
call   [eax]
popfd
ret
```

With Synthetic Instructions

```
vmpushfd
vmcli
mov    eax, [ebp+8]
call   [eax]
vmpopf
ret
```

All synthetic instructions trap and are five bytes long so they can be replaced with jmp or call instructions at runtime.

This sequence allows correct behavior in direct execution, but generates three traps.

© Microsoft Corporation 2004
Guest OS Patching

Runtime Guest OS Patching

– Replace synthetic instructions with subroutine calls
– This technique prevents us from exposing internal VMM implementation details to OS vendors. We can change the subroutine implementations in the future.

```
Original Code
pushfd
cli
mov eax,[ebp+8]
call [eax]
popfd
ret

With Synthetic Instructions
vmpushfd
vmcli
mov eax,[ebp+8]
call [eax]
vmpopf
ret

With Runtime Patches
call _vmpushfd
call _vmcli
mov eax,[ebp+8]
call [eax]
call _vmpopf
ret
```

This patched sequence is correct and fast
Direct Execution Overhead

Necessary to trap into the VMM kernel on some instructions

- IN & OUT for I/O device emulation
- STI & CLI for interrupt mask virtualization
- INT & IRET to catch ring transitions
- INVLPG and MOV to CR3 for page table virtualization

Traps are expensive – and getting worse

- ~500 cycles on Pentium III or AMD processors; ~2000 cycles on Pentium 4
- Runtime patching of some trapping instructions is possible
Physical Memory & RAM

Virtualized RAM
- User decides how much RAM is associated with each virtual machine

Physical pages
- Allocated by VMM from host OS
- Currently allocated at the time the VM starts, but could be allocated on demand
- Host physical addresses don’t match guest physical addresses
Logical Page Mappings

Logical Memory

- Logical mappings defined by guest page tables (mostly)
- VMM finds 32MB unused area for the VMM code and data (the “VMM work area”).
- VMM monitors guest OS address space usage and relocates itself if necessary
VMM Page Tables

VMM maintains its own private page table
- Initially, only the VMM work area is mapped

© Microsoft Corporation 2004
VMM Page Tables

VMM maintains its own private page table
- Initially, only the VMM work area is mapped
- Guest pages are mapped on demand as they are accessed
VMM Page Tables

VMM maintains its own private page table
- Initially, only the VMM work area is mapped
- Guest pages are mapped on demand as they are accessed
- Guest pages are unmapped when guest flushes its TLB
- VMM work area is relocated as necessary
Memory Sharing

Memory allocated with VMM APIs can be used in three ways

– Mapped within the VMM work area
– As guest virtual RAM (mapped into the guest address space according to the guest page tables)
– Mapped within the host context (for emulated DMA operations)
Device Emulation

Device emulation modules

- Emulate behaviors of a real hardware device
- Register “callbacks” for I/O port accesses
- Can access virtualized “RAM” for emulated DMA operations
- Communicate among themselves (e.g. Ethernet module “plugs into” the PCI bus module and communicates with the PIC module to assert interrupts)
- May call host services to perform emulation
- Can be suspended, saved and restored
Device I/O Accesses

I/O accesses (IN & OUT instructions)
- Trap into VMM kernel
- Force a context switch back to the host context where device emulation module is invoked
- “Fast I/O handlers” can be called from within the VMM context
- Some OUTs can be batched

MMIO accesses
- Caught in VMM’s page fault handler
- Very expensive
Discussion