Supporting Software

Compilers and Autotuners
Virtual Machines
Middleware
Operating Systems

Virtual Machines Outline

- Implementation
- Virtual machines on multiprocessors
  - Trango
  - Rts hypervisor
  - Georgiatech projects
- PVM
- DSM

Main features of VMM

- Paravirtualization and binary translation
- Memory management
  - Baloon process
- I/O management
  - Hosted mode

What is a VM?

A virtualized system that

- Provides a consistent ABI to guest programs
- Runs on a host system (software + hardware)
- Controls resources available to guest programs
- May provide different resources than hardware
  - Different Type (ex: JVML in Java VM)
  - Different Quantity (ex: more/fewer CPUs, disks, etc.)
- May be of two major types
  - **Process**: provides VM to a single process.
  - **System**: emulates an entire machine w/ guest OS.

Why use Virtual Machines?

Portability
- Run software on a different OS
- Run software on a different CPU

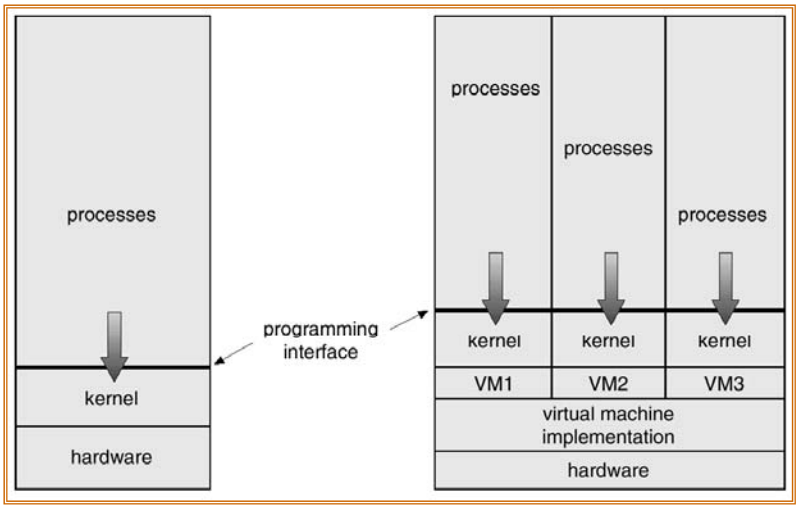
Aggregation
- Modern machines are fast and underused
- Put multiple servers in VMs on one real machine

Development
- Complex software environments
- Processor testing and simulation

Debugging
- Can analyze every aspect of hardware behavior

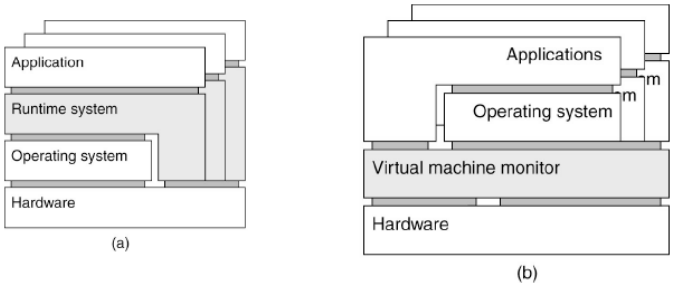
Security
- VMs provide greater isolation of software than regular OS

System Models



Types of Implementations

- A: Process VM:
  - Application-level virtualization
  - E.g. JVM
- B: System VM
  - VMM /hypervisor
  - E.g. Vmware ESX server (hardware support), XEN (paravirtualization), Vmware (binary translation)



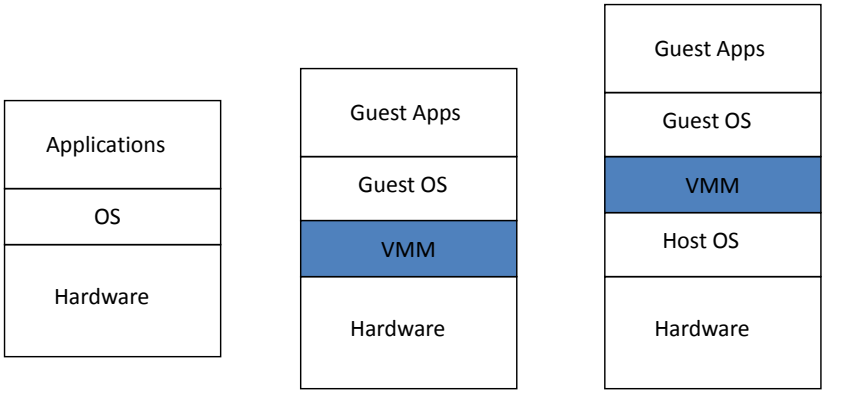
Process VMs

- Multitasking
  - Each process in a multitasking OS
  - VM = System call interface + ISA + VirtMem
- Emulators
  - Allow a process to run on a different OS/ISA
  - Types:
    - Interpreter
    - Dynamic binary translator
- High Level Language VMs
  - ex: Pascal, JVM, CLR

System VMs

- Virtual Machine Monitor (VMM)
  - Provides illusion of multiple isolated machines
  - Manages allocation of and access to hardware resources for multiple guest OSes
  - Layer between hardware and guest OS
- VMM tasks
  - State management
  - Resource control

System VMs



a. Traditional OS b. Native VMM c. User-mode Hosted VMM

Virtualizable Architecture Requirements

- ***Equivalence***:
  - Software on the VM executes identically to its execution on hardware, barring timing effects
- ***Performance***:
  - The vast majority of guest instructions are executed on the hardware without VMM intervention
- ***Safety***:
  - The VMM manages all hardware resources

Instruction Types

- ***Privileged:***
  - instructions are those that trap if the processor is in user mode and do not trap if it is in system mode
- ***Control sensitive:***
  - instructions are those that attempt to change the configuration of resources in the system
- ***Behavior sensitive:***
  - instructions are those whose behavior or result depends on the configuration of resources (the content of the relocation register or the processor's mode)

Virtualizable Architectures

- An architecture is ***virtualizable*** if the sets of behavior and control sensitive instructions are subsets of the set of privileged instructions
- On a virtualizable arch, a VMM works using a ***trap and emulate*** technique
  - Normal instructions run directly on processor
  - Privileged instructions trap into the VMM
  - The VMM emulates the effect of the privileged instructions for the guest OS

VMM Modes

- Safety: guest OS may not change hardware resources to impact other VMs or the VMM
- Guest OS runs in user mode
- VMM runs in supervisor mode
  - Tracks virtual mode of VM
  - User programs run in virtual user mode
  - OS runs in virtual supervisor mode
- Exceptions & interrupts invoke VMM
  - VMM can handle directly or produce a virtual exception for guest OS

System VM Execution

1. Timer Interrupt in running VM
2. Context switch to VMM
3. VMM saves state of running VM
4. VMM determines next VM to execute
5. VMM sets timer interrupt
6. VMM restores state of next VM
7. VMM sets PC to timer interrupt handler of next VM
8. Next VM active

Virtualizing Processor

- A CPU architecture is virtualizable if it supports direct execution:
  - Executing the VM on the real machine and the VMM retain ultimate control of the CPU
- All instructions that read or write privileged state trap when executed in guest OS
  - Some traps result from instruction type (I/O)
  - Other traps result from VMM protecting structures (memory pages)

Handling Privileged Instructions

1. Instruction Trap invokes VMM Dispatcher
2. Dispatcher calls Instruction Routine
3. Changes mode to supervisor
4. Emulates instruction
5. Computes return target
6. Restores mode to user
7. Jumps to target

x86 is not virtualizable

- x86 architecture is not virtualizable:
  - 17 sensitive non-privileged instructions.
- **Visibility of privileged state**:
  - Guest OS can observe that current privilege level (CPS) in code segment selection (%cs) is not kernel.
- **Lack of traps when privileged instructions run at user level**:
  - Certain instructions act differently in kernel mode than user mode, but don't cause a trap in user mode so the VMM can detect this

Example x86 Problem: POPF

- POPF instruction
  - Pops flag registers from stack
  - Includes interrupt-enable flag
  - User mode, POPF modifies all but interrupt flag
  - Kernel mode, POPF modifies all flags

Solutions

- Paravirtualization
  - Patch source code containing problematic instructions
- Binary translation
  - Patch binary code the first time it executes
  - Can be applied to emulate a different ISA

Dynamic Binary Translation

Translate machine code at runtime.

- Often x86 to x86 translation, but
- Apple uses for emulating older processors.

VM interleaves translation and execution

1. Translate basic block (BB) of code.
2. Execute translated BB'.
3. Transfer control to next BB.
4. If next BB already translated, execute it.
5. Otherwise goto 1.

C Code Example

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

Assembly Version

```
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
```

Basic Block Translation

- Most instructions copied identically.
- Privileged instructions must be emulated.
- Jumps must be translated since translation can alter code layout.
- Each translated BB must end with jump to next translated BB.

```
isPrime:  mov %ecx, %edi
          mov %esi, $2
          cmp %esi, %ecx
          jge prime
```

```
isPrime':  mov %ecx, %edi    ; IDENT
           mov %esi, $2
           cmp %esi, %ecx
           jge [takenAddr]   ; JCC
           jmp [fallthrAddr]
```

Translation of `isPrime(49)`

Note that prime: BB never translated since 49 is not prime.

```
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:0xff39eb8(%rip), %rcx  ; spill %rcx
            movzx %ecx, %r11b
            jmp   %gs:0xfc7dde0(8*%rcx)
```

Intel VT Extensions

- Intel VT allows trap and emulate VMM on newer x86 chips.
- VMCB
  - Virtual Machine Control Block
  - Control state + subset of guest VM state
- Guest mode
  - New less privileged execution mode to allow direct execution of guest code.
- vmrun
  - New instruction to transfer from host mode to guest mode.
  - Guest execution proceeds until condition specified in VMCB met, at which point hardware performs an exit operation, saving guest state to VMCB and loading VMM state, then executing VMM in host mode.

Intel VT Extensions

- Instructions
  - Some sensitive instructions operate on non-root VMX state; others produce a VM exit.
  - VMCB controls which instructions VM exit.
- Interrupts
  - External interrupts cause VM exits.
  - VMCB controls which exceptions VM exit.

VMWare

- x86 dynamic binary translation VM
  - Direct execution in user mode
  - Binary translation in kernel mode
- VMWare Workstation, Player, Server
  - Hosted VMM runs on Linux or Windows
  - Any x86 OS can be used as guest OS
- VMWare ESX Server
  - Native VMM runs directly on x86 hardware
  - VMotion allows VM migration

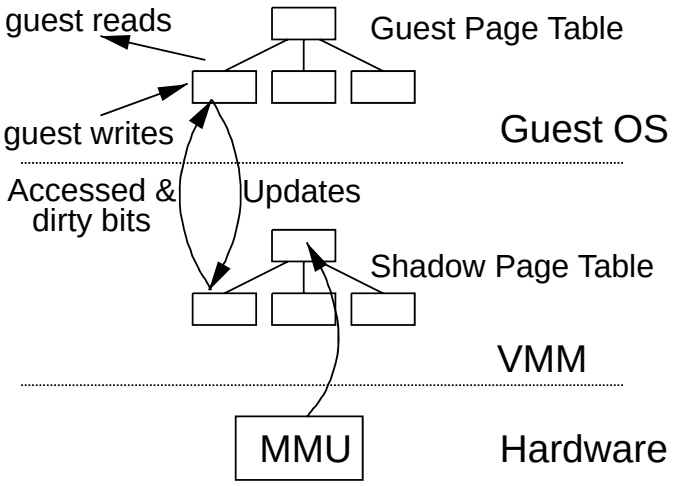
Virtualizing Memory

- ***Virtual Memory***: Each process has its own page table managed by the guest OS pointing to real memory of the VM its running in
- ***Real Memory***: Memory allocated to each VM by the VMM. It is mapped to the physical memory of the host hardware
- ***Physical Memory***: The physical memory of the host hardware

Shadow Page Tables

- Guest OS maintains its own page tables
  - Virtual to real memory mapping
- VMM maintains ***shadow page tables***
  - Virtual to physical memory mapping
  - Used by hardware to translate virtual addresses
  - VMM validates guest page table updates
  - Replicates guest changes in shadow page table
- Virtualize page table pointer register
  - VMM manages real page table pointer
  - Updates page table ptr when switching VMs

Shadow Page Tables



Memory Management Issues

- VMM can page out some portions of the VM
- However, is the guest OS that knows better what are processes requirements
- Vmware ESX server solution:
  - Baloon process running inside the guest OS and communicating with VMM
  - VMM inflates the baloon when it wants to get more memory from that VM: exploits guestOS page replacement strategies

Memory Management Issues

- Redundant copies of data and code between different VMs
- Solution:
  - Content based page sharing

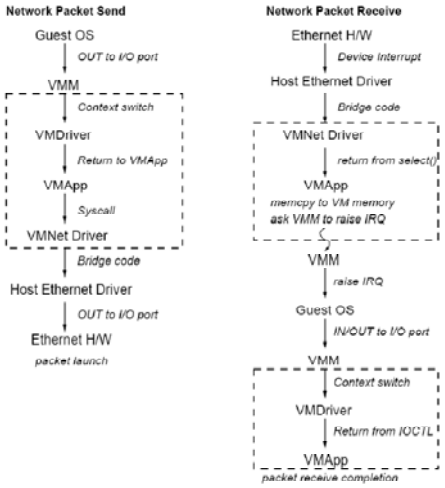
Virtualizing I/O

- VMM must intercept all guest I/O ops
  - PC: privileged IN and OUT instructions
  - I/O operation may consist of many INs/OUTs
- Problem: huge array of diverse hardware
  - Native VMM needs driver for each device
  - Hosted VMM uses host drivers w/ perf penalty

Virtualizing Devices

- Dedicated Devices
  - VM has sole control of device
- Partitioned Devices
  - VM has dedicated slice of device, treats as full
  - VMM translates virtual full dev parameters to parameters for underlying physical device.
- Shared Devices
  - VMM can multiplex devices.
  - Each VM may have own virtual device state.
- Nonexistent Devices
  - Virtual software devices with no physical counterpart.

Virtualizing a Network Card



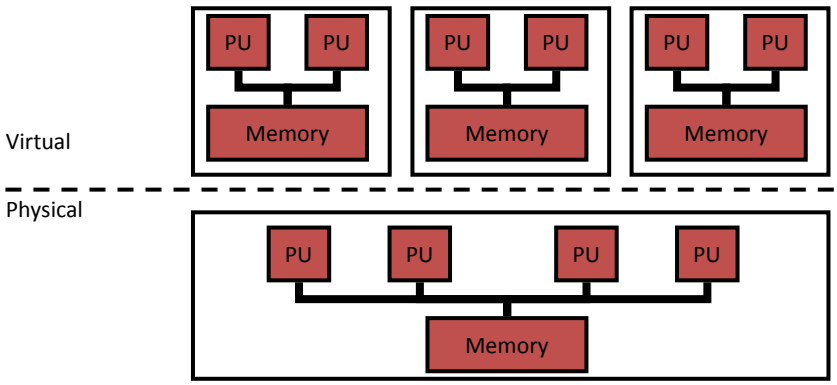
VM Performance

- Why is VM slower than physical hardware?
  - **Emulation**: Sensitive instructions must be emulated
  - **Interrupt Handling**: VMM must handle interrupts, even if eventually passed to guest
  - **Context Switches**: VMM must save VM state when controlled transferred to VMM
  - **Bookkeeping**: VMM has to do work to simulate behavior of real machine, such as keeping track of time for VMs
  - **Memory**: Memory accesses may require access to both shadow and local page tables

VM on Multiprocessors

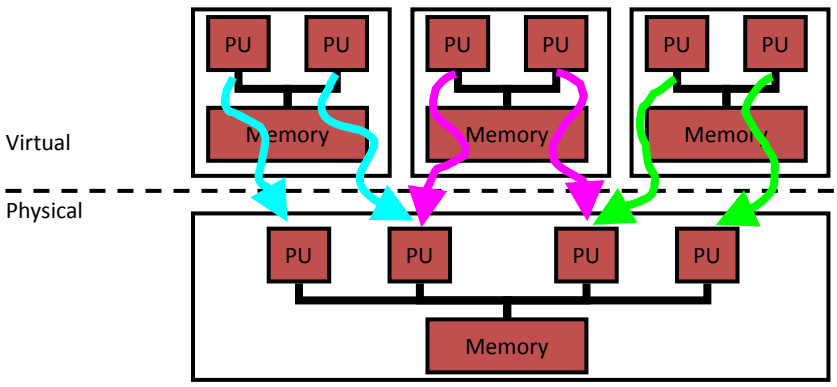
- Benefits:
  - Distribution of physical processors to guest Oses
  - Reassignment depending on workload conditions
    - VM migration
- Issues:
  - Lock-holder preemption avoidance
  - Sub-optimal scheduling

Multiprocessor Virtual Machine



Multiprocessor Virtual Machine

- Multiplexing multiple virtual CPUs on one physical CPU



40

Lock-Holder Preemption Problem

- Critical sections are used in SMP kernels to ensure in-order data structure updates
  - Fine grained locking to prevent performance hit and preserve scalability
- The VMM can preempt a lock holder, extending lock holding time
  - Violation of statistical fairness of the lock

Solutions

- Co-scheduling:
  - Each virtual cpu on a physical cpu for an equal time slice
  - Even if a lock holder is preempted another processor will not spin on the preempted lock
- Problems
  - Scalability
  - Flexibility
  - Cannot multiplex virtual CPU on physical CPU for fault recovery or load balancing

Intrusive Lock Holder Preemption Avoidance

- Modify guest OS to give hints to VMM
- Before acquiring a lock, OS indicates that it cannot be preempted for the next n microseconds

Non-Intrusive Lock Holder Preemption Avoidance

- VMM monitors switches between user-level and kernel-level modes
  - Determine safe and unsafe states
- Safe state: user level
  - No lock holder possible
- Unsafe state: kernel level
  - Lock hold possible
  - Monitor IA-32 HALT instruction to know if processor is doing the idle loop (safe state)

Locking-Aware VM Scheduling

- VCPUs are as threads for virtualization layer that must be scheduled
- Scheduling properties:
  - Fair access of VM to physical CPUs
  - Lock-holder preemption avoidance

Time Balooning

- Problem: a multiprocessor OS bases load-balancing decisions on physical CPU parameters that are modified by the VMM
  - Wrong process distribution among virtual processors
  - Need more information about physical resource allocation
- Solution
  - Insert a baloon module that polls the VMM to know about processing time for that VM
  - When no physical cpu is allocated, it inflates the baloon (generating virtual workload) to correct scheduler assumption