Module 5: Deadlocks

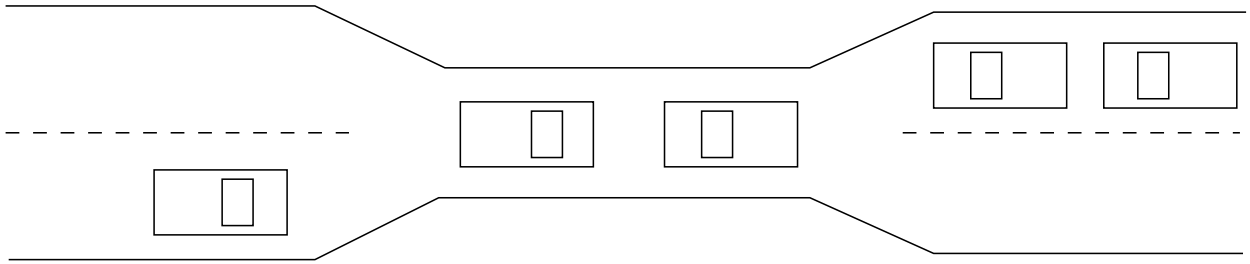
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - semaphores A and B, initialized to 1

P_0	P_1
wait(A);	*wait*(B)
wait(B);	*wait*(A)

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

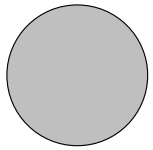
Resource-Allocation Graph

A set of vertices V and a set of edges E.

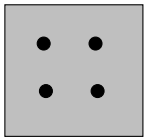
- V is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.
- *request edge* – directed edge $P_i \rightarrow R_j$
- *assignment edge* – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

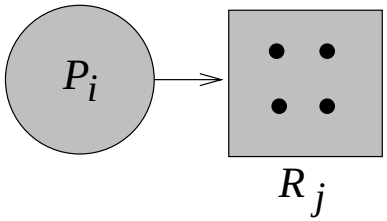
- Process



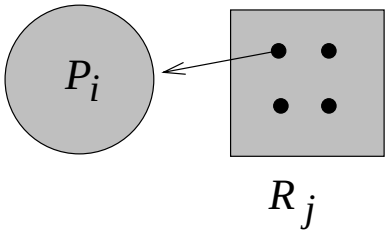
- Resource type with 4 instances



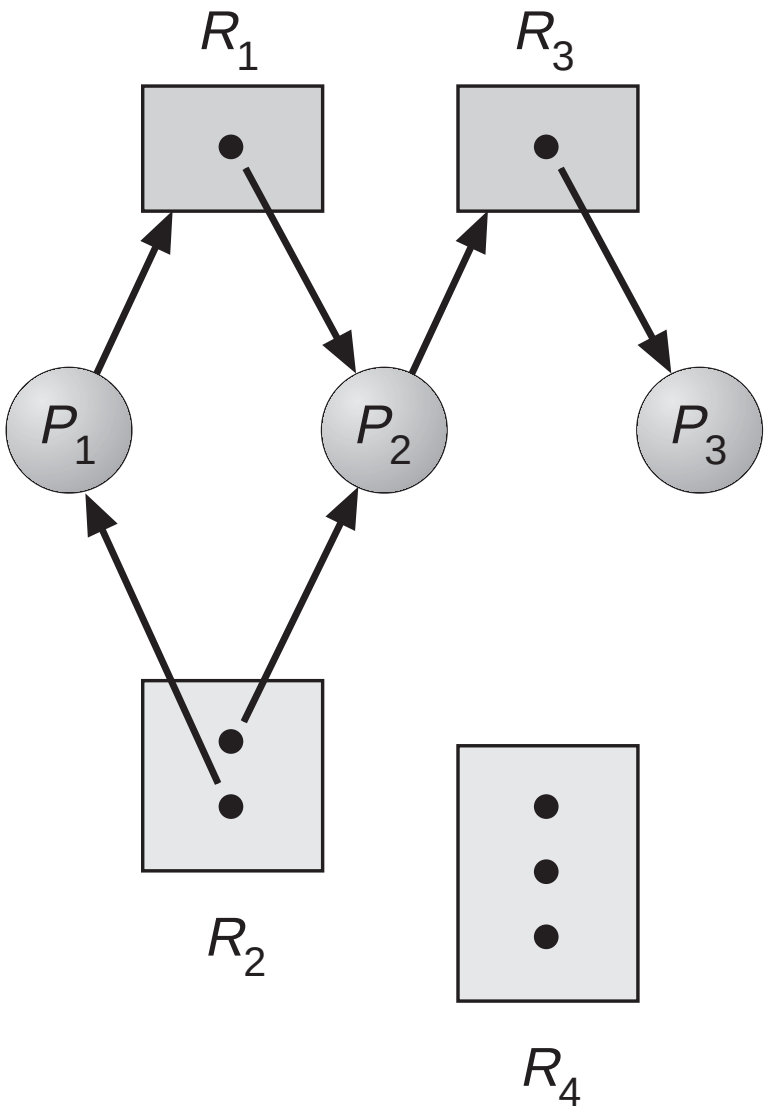
- P_i requests instance of R_j



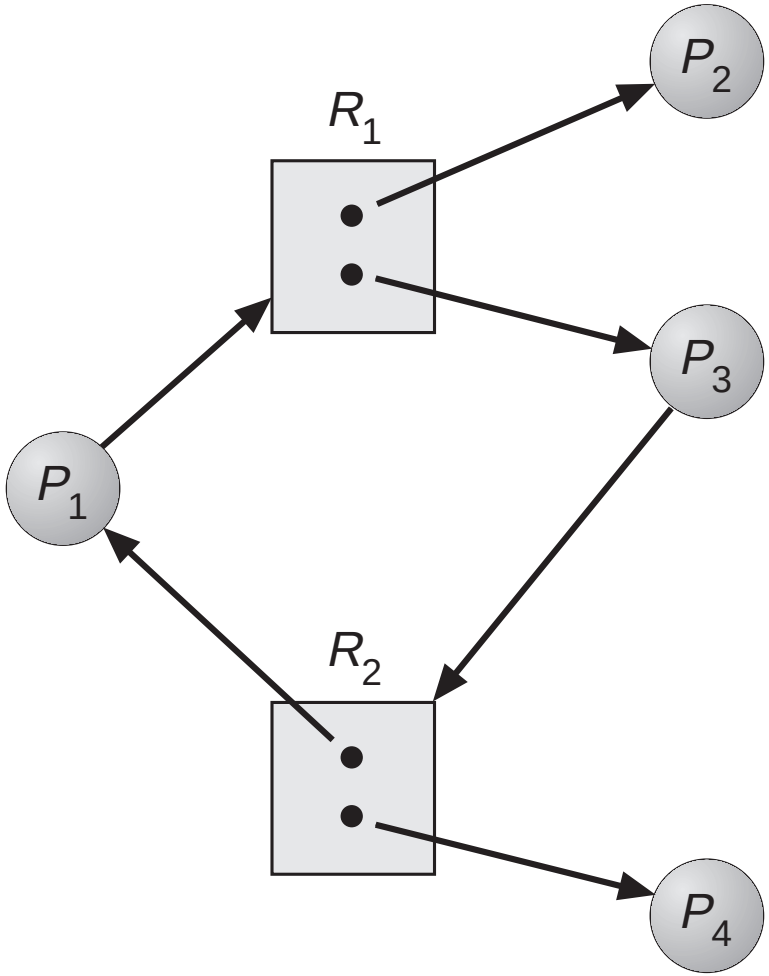
- P_i is holding an instance of R_j



Example of a Graph With No Cycles



Example of a Graph With a Cycle



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Restrain the ways resource requests can be made.

- Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources.
- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- No Preemption –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a *safe sequence* of all processes.
- Sequence $<P_1, P_2, ..., P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Silberschatz and Galvin ©1998

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Silberschatz and Galvin ©1998

Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Silberschatz and Galvin ©1998

Banker's Algorithm

- Multiple instances.
- Each process must *a priori* claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Silberschatz and Galvin ©1998

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resource types.

- *Available:* Vector of length m. If *Available*[j] = k, there are k instances of resource type R_j available.
- *Max:* $n \times m$ matrix. If *Max*[i,j] = k, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* $n \times m$ matrix. If *Allocation*[i,j] = k, then P_i is currently allocated k instances of R_j.
- *Need:* $n \times m$ matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task.

$$Need[i,j] = Max[i,j] - Allocation[i,j].$$

Silberschatz and Galvin ©1998

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively. Initialize:

   $Work := Available$

   $Finish[i] := false$ for $i = 1, 2, ..., n.$

2. Find an *i* such that both:

   (a) $Finish[i] = false$

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

3. $Work := Work + Allocation_i$

   $Finish[i] := true$

   go to step 2.

4. If $Finish[i] = true$ for all *i,* then the system is in a safe state.

Silberschatz and Galvin ©1998

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$, then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available := Available - Request_i;$$
$$Allocation_i := Allocation_i + Request_i;$$
$$Need_i := Need_i - Request_i;$$

   - If safe $\Rightarrow$ the resources are allocated to P_i.
   - If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored.

Silberschatz and Galvin ©1998

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types A (10 instances), B (5 instances), and C (7 instances).
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Silberschatz and Galvin ©1998

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max* − *Allocation*.

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria.

Silberschatz and Galvin ©1998

Example (Cont.): P_1 requests (1,0,2)

- Check that *Request* $\leq$ *Available* (that is, (1,0,2) $\leq$ (3,3,2)) $\Rightarrow$ true.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Silberschatz and Galvin ©1998

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Silberschatz and Galvin ©1998

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Silberschatz and Galvin ©1998

Several Instances of a Resource Type

Data Structures

- *Available:* A vector of length m indicates the number of available resources of each type.
- *Allocation:* An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- *Request:* An $n \times m$ matrix indicates the current request of each process. If $Request[i,j] = k$, then process P_i is requesting k more instances of resource type R_j.

Silberschatz and Galvin ©1998

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively. Initialize:

   (a) *Work* := *Available.*

   (b) For $i = 1, 2, ..., n,$ if $Allocation_i \neq 0$, then *Finish*[*i*] := *false;* otherwise, *Finish*[*i*] := *true.*

2. Find an index *i* such that both:

   (a) *Finish*[*i*] = *false.*

   (b) $Request_i \leq Work.$

   If no such *i* exists, go to step 4.

Silberschatz and Galvin ©1998

Detection Algorithm (Cont.)

3. *Work* := *Work* + *Allocation*$_i$
   Finish[*i*] := *true*
   go to step 2.

4. If *Finish*[*i*] = false, for some *i,* $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if *Finish*[*i*] = *false,* then P_i is deadlocked.

Algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Silberschatz and Galvin ©1998

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i.*

Silberschatz and Galvin ©1998

Example (Cont.)

- P_2 requests an additional instance of type *C.*

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes' requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Silberschatz and Galvin ©1998

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Silberschatz and Galvin ©1998

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Silberschatz and Galvin ©1998

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process from that state.
- Starvation – same process may always be picked as victim; include number of rollback in cost factor.

Silberschatz and Galvin ©1998

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

  allowing the use of the optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.

Silberschatz and Galvin ©1998

Unit 5

VIRTUAL MACHINES

1. Benefits and Features
2. Building Blocks
3. Types of Virtual Machines and their Implementations
4. Virtualization and Operating-System Components

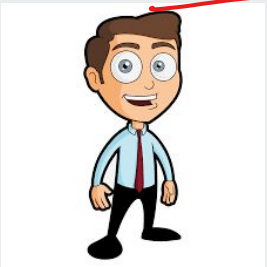
Room ⇒ H/W (Host)

Single ⇒ OS - Windows

H/W ← Windows, Linux

4 sharing

VMM



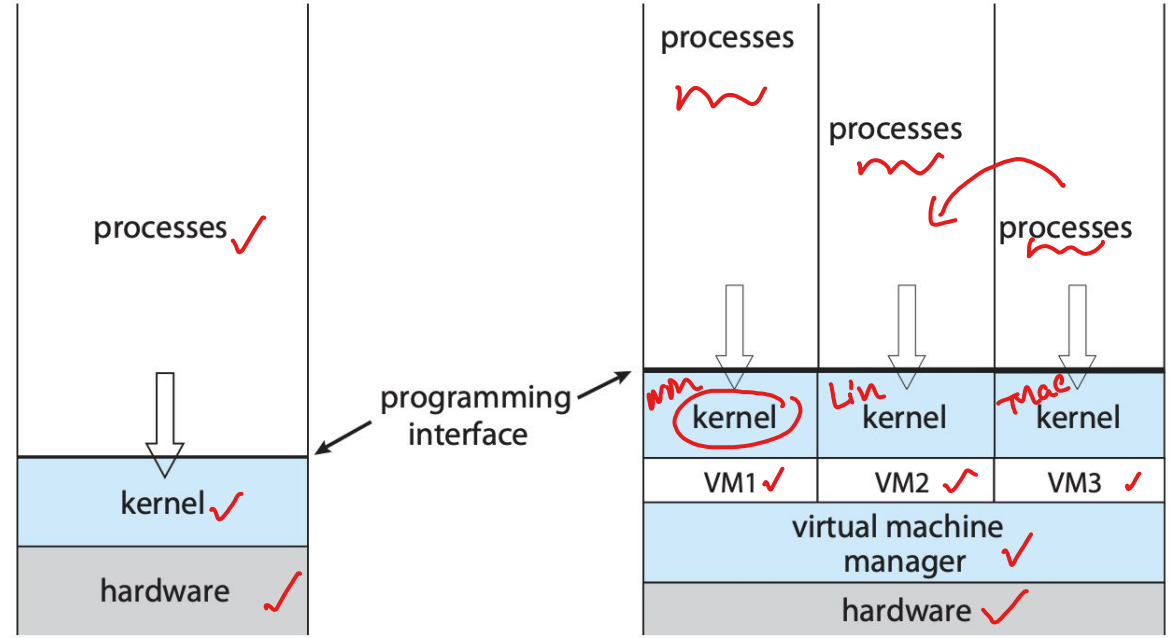
Win Linux

VM1 VM2 VM3

VMM

H/W

System Models



Non-virtual machine

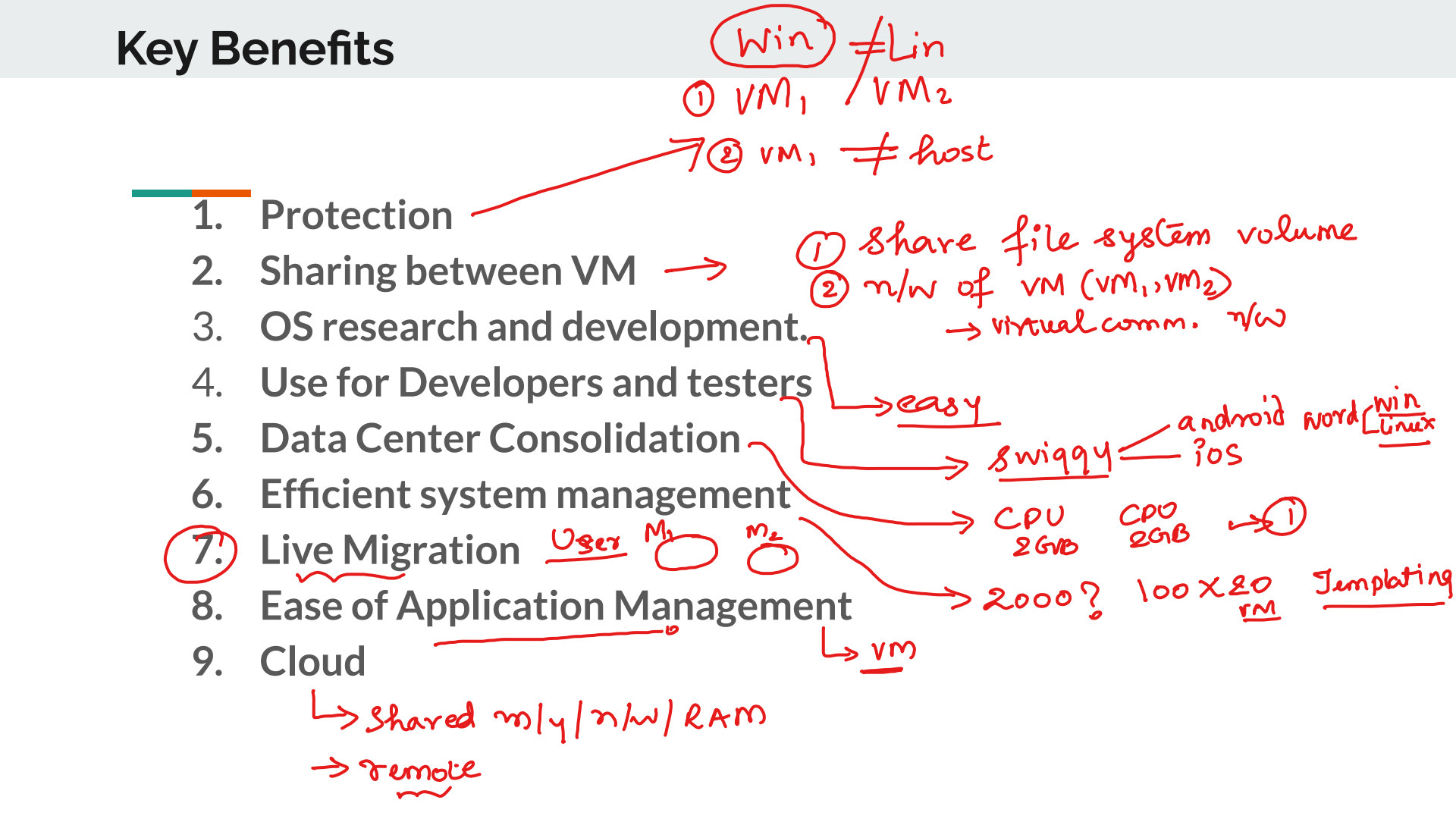
Virtual machine

Overview

- Fundamental idea – abstract hardware of a single computer into several different execution environments
  - Similar to layered approach
  - But layer creates virtual system (**virtual machine**, or **VM**) on which operation systems or applications can run
- Several components
  - **Host** – underlying hardware system
  - **Virtual machine manager** (**VMM**) or **hypervisor** – creates and runs virtual machines by providing interface that is ***identical*** to the host
    - (Except in the case of paravirtualization)
  - **Guest** – process provided with virtual copy of the host
    - Usually an operating system
- Single physical machine can run multiple operating systems concurrently, each in its own virtual machine

Key Benefits

1. Protection
2. Sharing between VM
3. OS research and development
4. Use for Developers and testers
5. Data Center Consolidation
6. Efficient system management
7. Live Migration
8. Ease of Application Management
9. Cloud



Benefits points

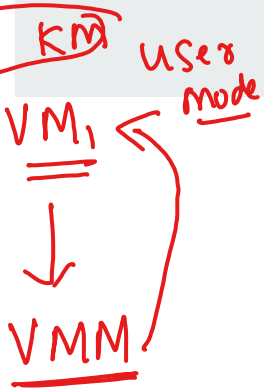
1. **Protection**
    a. Host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests as each virtual machine is almost completely isolated from all other virtual machines, there are almost no protection problems.
2. **Sharing between VM**
    a. First, it is possible to share a file-system volume and thus to share files. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. The network is modeled after physical communication networks but is implemented in software
    b. ability to freeze, or suspend, a running virtual machine. VMM allow copies and snapshots to be made of the guest. The copy can be used to create a new VM or to move a VM from one machine to another with its current state intact. The guest can then resume where it was, as if on its original machine, creating a clone.
3. **OS research and development.**
    a. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and a change in one part may cause obscure bugs to appear in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.
    b. VM's can be used to deploy and test bugs in changes made to OS.

4. **Use for Developers and testers**
    a. Multiple virtualized workstation allows for rapid porting and testing of programs in varying environments. In addition, multiple versions of a program can run, each in its own isolated operating system, within one system.
5. **Data Center Consolidation**
    a. It involves taking two or more separate systems and running them in virtual machines on one system. Such physical-to-virtual conversions result in resource optimization, since many lightly used systems can be combined to create one more heavily used system.
6. **Efficient system management**
    a. management tools that are part of the VMM allow system administrators to manage many more systems than they otherwise could. A virtual environment might include 100 physical servers, each running 20 virtual servers. Without virtualization, 2,000 servers would require several system administrators. With virtualization and its tools, the same work can be managed by one or two administrators.
    b. One of the tools that make this possible is templating, in which one standard virtual machine image, including an installed and configured guest operating system and applications, is saved and used as a source for multiple running VMs.
7. **Live Migration**
    a. Virtualization can improve not only resource utilization but also resource management. Some VMMs include a live migration feature that moves a running guest from one physical server to another without interrupting its operation or active network connections. If a server is overloaded, live migration can thus free resources on the source host while not disrupting the guest
8. **Ease of Application Management**
    a. the application could be preinstalled on a tuned and customized operating system in a virtual machine. This method would offer several benefits for application developers. Application management would become easier, less tuning would be required, and technical support of the application would be more straightforward.
9. **Cloud**
    a. Cloud computing, for example, is made possible by virtualization in which resources such as CPU, memory, and I/O are provided as services to customers using Internet technologies. By using APIs, a program can tell a cloud computing facility to create thousands of VMs, all running a specific guest operating system and application, that others can access via the Internet.

Building Blocks

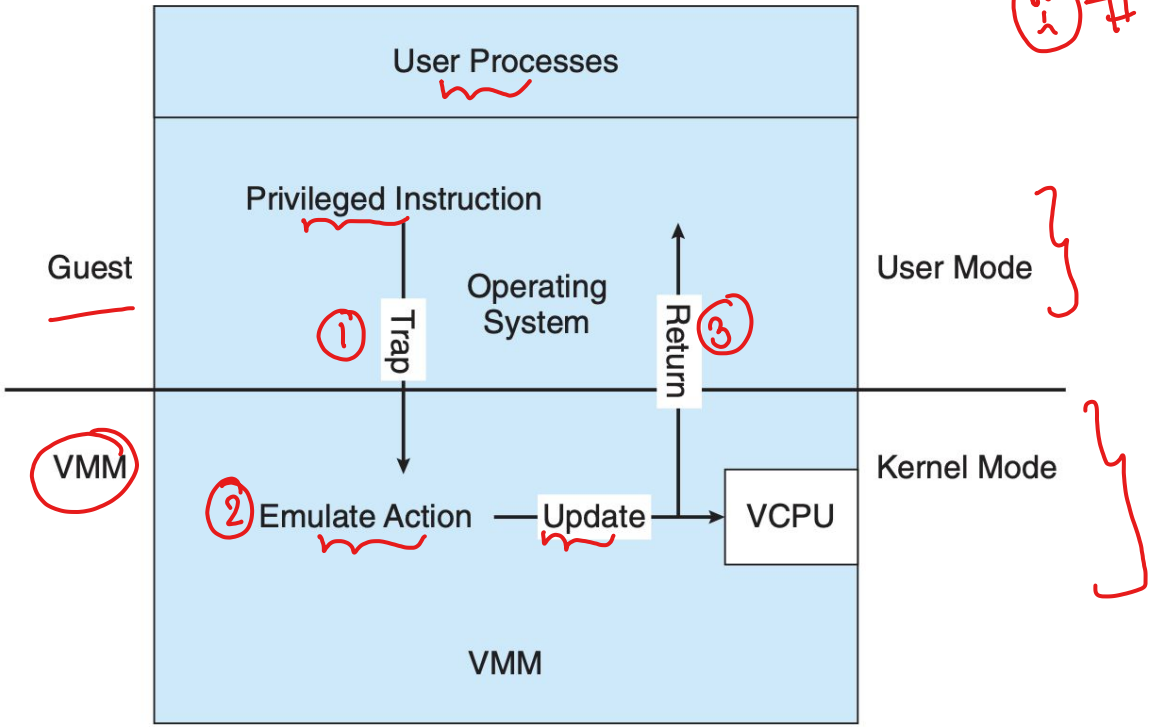
- Generally difficult to provide an ***exact*** duplicate of underlying machine
  - Especially if only dual-mode operation available on CPU
  - But getting easier over time as CPU features and support for VMM improves
  - Most VMMs implement **virtual CPU** (**VCPU**) to represent state of CPU per guest as guest believes it to be
    - When guest context switched onto CPU by VMM, information from VCPU loaded and stored
  - Several techniques, as described in next slides

Building Block – Trap and Emulate



- Dual mode CPU means guest executes in user mode
  - Kernel runs in kernel mode
  - Not safe to let guest kernel run in kernel mode too
    - So VM needs two modes – virtual user mode and virtual kernel mode
      - Both of which run in real user mode
    - Actions in guest that usually cause switch to kernel mode must cause switch to virtual kernel mode

Trap-and-Emulate Virtualization Implementation



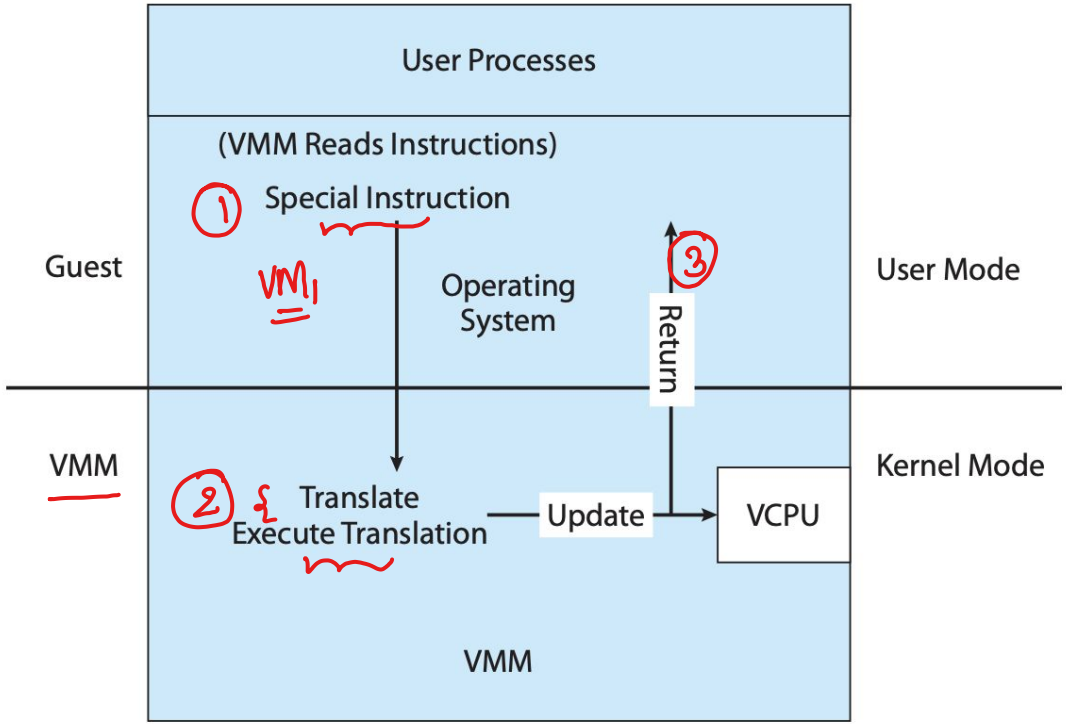
Trap-and-Emulate (Cont.)

- How does switch from virtual user mode to virtual kernel mode occur?
  - Attempting a privileged instruction in user mode causes an error -> trap
  - VMM gains control, analyzes error, executes operation as attempted by guest
  - Returns control to guest in user mode
  - Known as **trap-and-emulate**
  - Most virtualization products use this at least in part
- User mode code in guest runs at same speed as if not a guest
- But kernel mode privilege mode code runs slower due to trap-and-emulate
  - Especially a problem when multiple guests running, each needing trap-and-emulate
- CPUs adding hardware support, mode CPU modes to improve virtualization performance

Building Block – Binary Translation

- Some CPUs don't have clean separation between privileged and nonprivileged instructions
  - Earlier Intel x86 CPUs are among them
    - Earliest Intel CPU designed for a calculator
  - Backward compatibility means difficult to improve
  - Consider Intel x86 `popf` instruction
    - Loads CPU flags register from contents of the stack
    - If CPU in privileged mode -> all flags replaced
    - If CPU in user mode -> only some flags replaced
      - No trap is generated

Binary Translation Virtualization Implementation



Binary Translation (Cont.)

Other similar problem instructions we will call ***special instructions***

- Caused trap-and-emulate method considered impossible until 1998

Binary translation solves the problem

1. Basics are simple, but implementation very complex
2. If guest VCPU is in user mode, guest can run instructions natively
3. If guest VCPU in kernel mode (guest believes it is in kernel mode)
   a) VMM examines every instruction guest is about to execute by reading a few instructions ahead of program counter
   b) Non-special-instructions run natively
   c) Special instructions translated into new set of instructions that perform equivalent task (for example changing the flags in the VCPU)

Binary Translation (Cont.)

- Implemented by translation of code within VMM
- Code reads native instructions dynamically from guest, on demand, generates native binary code that executes in place of original code
- Performance of this method would be poor without optimizations
  - Products like VMware use caching
    - Translate once, and when guest executes code containing special instruction cached translation used instead of translating again
    - Testing showed booting Windows XP as guest caused 950,000 translations, at 3 microseconds each, or 3 second (5 %) slowdown over native

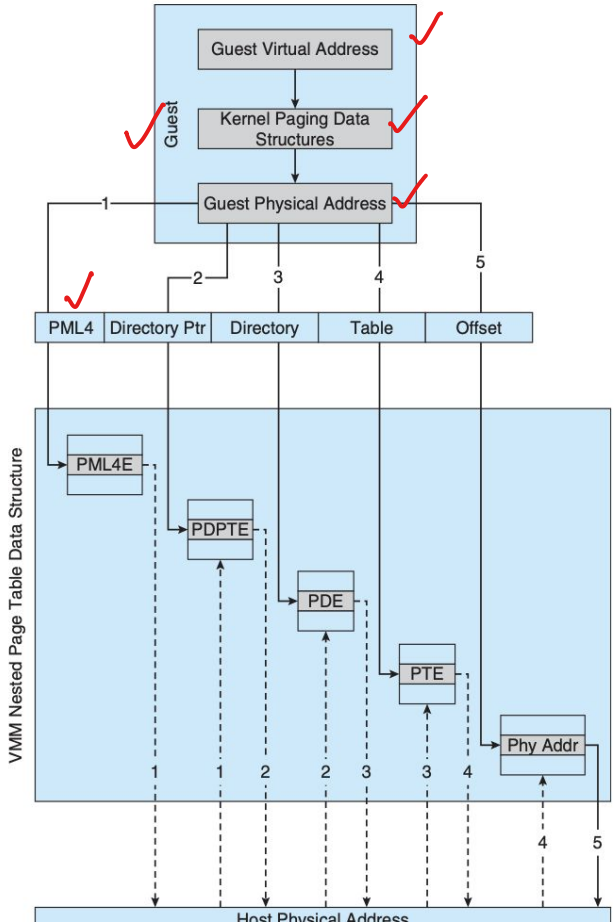
Nested Page Tables

- Memory management another general challenge to VMM implementations
- How can VMM keep page-table state for both guests believing they control the page tables and VMM that does control the tables?
- Common method (for trap-and-emulate and binary translation) is **nested page tables** (**NPTs**)
  - Each guest maintains page tables to translate virtual to physical addresses
  - VMM maintains per guest NPTs to represent guest's page-table state
    - Just as VCPU stores guest CPU state
  - When guest on CPU -> VMM makes that guest's NPTs the active system page tables
  - Guest tries to change page table -> VMM makes equivalent change to NPTs and its own page tables
  - Can cause many more TLB misses -> much slower performance

Building Blocks – Hardware Assistance

- All virtualization needs some HW support
- More support -> more feature rich, stable, better performance of guests
- Intel added new **VT-x** instructions in 2005 and AMD the **AMD-V** instructions in 2006
  - CPUs with these instructions remove need for binary translation
  - Generally define more CPU modes – "guest" and "host"
  - VMM can enable host mode, define characteristics of each guest VM, switch to guest mode and guest(s) on CPU(s)
  - In guest mode, guest OS thinks it is running natively, sees devices (as defined by VMM for that guest)
    - Access to virtualized device, priv instructions cause trap to VMM
    - CPU maintains VCPU, context switches it as needed
- HW support for Nested Page Tables, DMA, interrupts as well over time

Nested Page Tables



Types of VM

1. Type 0
2. Type 1
3. Type 2
4. Para
5. Programming environment
6. Emulation
7. Container

Life cycle

1) Creation
   - m/y
   - CPU
   - n/w
   - storage

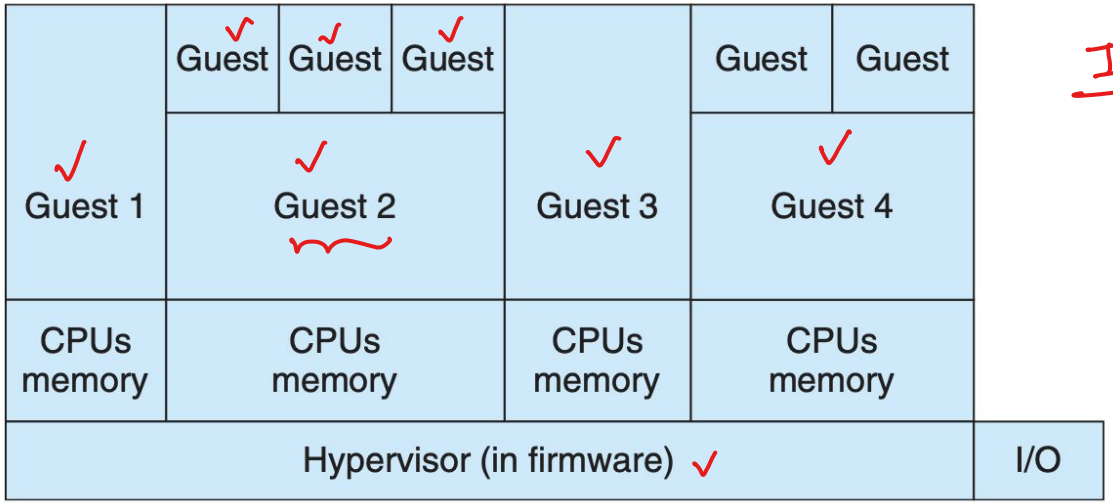
2) deleted (free space)

① Type 0 Hypervisor

— partitions → H/W

→ boot time

→ partition



I/O.

- Shared access
- Control partition

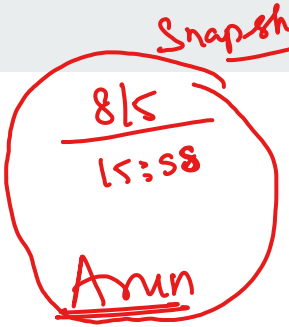
Types of VMs – Type 0 Hypervisor

- Old idea, under many names by HW manufacturers
  - "partitions", "domains"
  - A HW feature implemented by firmware
  - OS need to nothing special, VMM is in firmware
  - Smaller feature set than other types
  - Each guest has dedicated HW
- I/O a challenge as difficult to have enough devices, controllers to dedicate to each guest
- Sometimes VMM implements a **control partition** running daemons that other guests communicate with for shared I/O
- Can provide virtualization-within-virtualization (guest itself can be a VMM with guests
  - Other types have difficulty doing this

Types of VMs – Type 1 Hypervisor

- Commonly found in company datacenters
  - In a sense becoming "datacenter operating systems"
    - Datacenter managers control and manage OSes in new, sophisticated ways by controlling the Type 1 hypervisor
    - Consolidation of multiple OSes and apps onto less HW
    - Move guests between systems to balance performance
    - Snapshots and cloning

- Special purpose operating systems that run natively on HW
  - Rather than providing system call interface, create run and manage guest OSes
  - Can run on Type 0 hypervisors but not on other Type 1s
  - Run in kernel mode
  - Guests generally don't know they are running in a VM
  - Implement device drivers for host HW because no other component can
  - Also provide other traditional OS services like CPU and memory management



Types of VMs – Type 1 Hypervisor (Cont.)

- Another variation is a general purpose OS that also provides VMM functionality
  - RedHat Enterprise Linux with KVM, Windows with Hyper-V, Oracle Solaris
  - Perform normal duties as well as VMM duties
  - Typically less feature rich than dedicated Type 1 hypervisors
- In many ways, treat guests OSes as just another process
  - Albeit with special handling when guest tries to execute special instructions

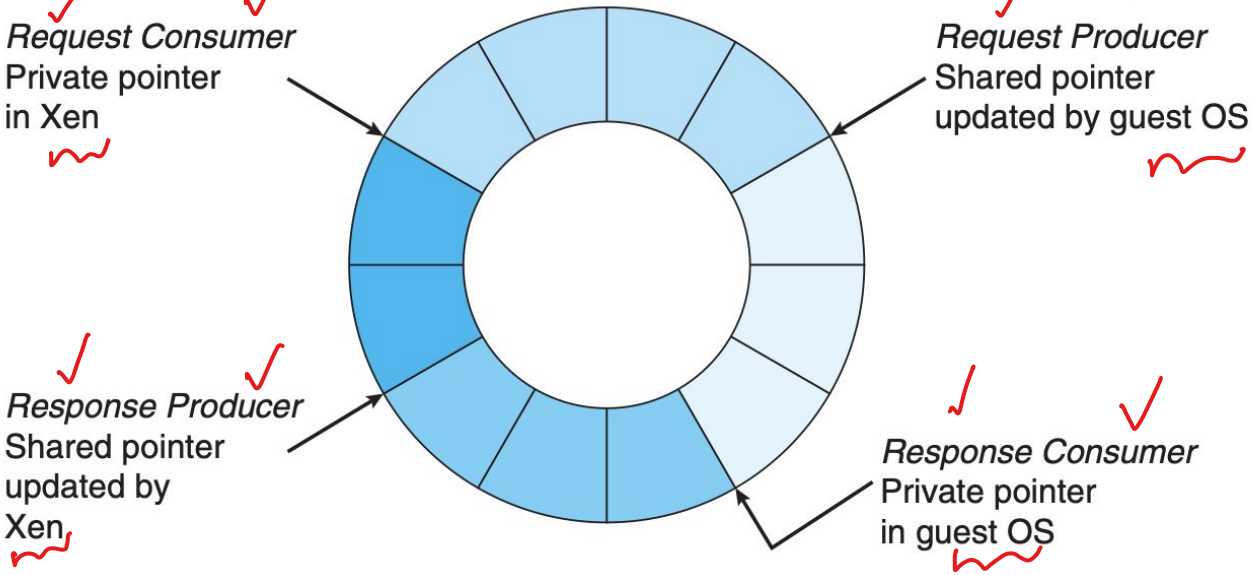
Types of VMs – Type 2 Hypervisor

- Less interesting from an OS perspective
  - Very little OS involvement in virtualization
  - VMM is simply another process, run and managed by host
    - Even the host doesn't know they are a VMM running guests
  - Tend to have poorer overall performance because can't take advantage of some HW features
  - But also a benefit because require no changes to host OS
    - Student could have Type 2 hypervisor on native host, run multiple guests, all on standard host OS such as Windows, Linux, MacOS

Types of VMs – Paravirtualization

- Does not fit the definition of virtualization – VMM not presenting an exact duplication of underlying hardware
  - But still useful!
  - VMM provides services that guest must be modified to use
  - Leads to increased performance
  - Less needed as hardware support for VMs grows
- Xen, leader in paravirtualized space, adds several techniques
  - For example, clean and simple device abstractions
    - Efficient I/O
    - Good communication between guest and VMM about device I/O
    - Each device has circular buffer shared by guest and VMM via shared memory

Xen I/O via Shared Circular Buffer



Request queue - Descriptors queued by the VM but not yet accepted by Xen

Outstanding descriptors - Descriptor slots awaiting a response from Xen

Response queue - Descriptors returned by Xen in response to serviced requests

Unused descriptors

Types of VMs – Paravirtualization (Cont.)

- Xen, leader in paravirtualized space, adds several techniques (Cont.)
  - Memory management does not include nested page tables
    - Each guest has own read-only tables
    - Guest uses **hypercall** (call to hypervisor) when page-table changes needed
- Paravirtualization allowed virtualization of older x86 CPUs (and others) without binary translation
- Guest had to be modified to use run on paravirtualized VMM
- But on modern CPUs Xen no longer requires guest modification -> no longer paravirtualization

Types of VMs – Programming Environment Virtualization

JVM

- Also not-really-virtualization but using same techniques, providing similar features
- Programming language is designed to run within custom-built virtualized environment
  - For example Oracle Java has many features that depend on running in **Java Virtual Machine** (**JVM**)
- In this case virtualization is defined as providing APIs that define a set of features made available to a language and programs written in that language to provide an improved execution environment
- JVM compiled to run on many systems (including some smart phones even)
- Programs written in Java run in the JVM no matter the underlying system
- Similar to **interpreted languages**

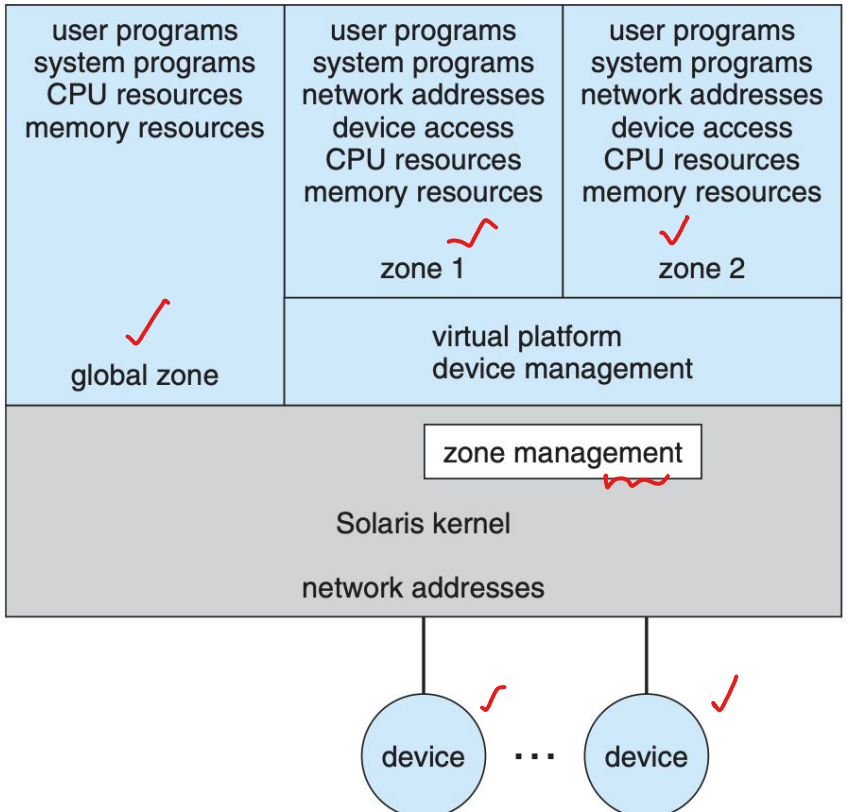
Types of VMs – Emulation

- Another (older) way for running one operating system on a different operating system
  - Virtualization requires underlying CPU to be same as guest was compiled for
  - Emulation allows guest to run on different CPU
- Necessary to translate all guest instructions from guest CPU to native CPU
  - Emulation, not virtualization
- Useful when host system has one architecture, guest compiled for other architecture
  - Company replacing outdated servers with new servers containing different CPU architecture, but still want to run old applications
- Performance challenge – order of magnitude slower than native code
  - New machines faster than older machines so can reduce slowdown
- Very popular – especially in gaming where old consoles emulated on new

Types of VMs – Application Containment

- Some goals of virtualization are segregation of apps, performance and resource management, easy start, stop, move, and management of them
- Can do those things without full-fledged virtualization
  - If applications compiled for the host operating system, don't need full virtualization to meet these goals
- Oracle **containers** / **zones** for example create virtual layer between OS and apps
  - Only one kernel running – host OS
  - OS and devices are virtualized, providing resources within zone with impression that they are only processes on system
  - Each zone has its own applications; networking stack, addresses, and ports; user accounts, etc
  - CPU and memory resources divided between zones
    - Zone can have its own scheduler to use those resources

Solaris 10 with Two Zones



Virtualization and Operating-System Components

- Now look at operating system aspects of virtualization
  - CPU scheduling, memory management, I/O, storage, and unique VM migration feature
    - How do VMMs schedule CPU use when guests believe they have dedicated CPUs?
    - How can memory management work when many guests require large amounts of memory?

① CPU SCHEDULING

OS

CPU

4

VM_1 VM_2 ... VM_{20} ⇒ Guests

VMM

CPU ⇒ 10 (phy)

① VM_1 4 < 10 ⇒ dedicated

C_1, C_2, C_3, C_4

② VM_3 — 30 [VCPU]

VMM (illusion)

→ portion

→ Time of day

∵ appln

CPU SCHEDULING

- The virtualization software presents one or more virtual CPUs to each of the virtual machines running on the system and then schedules the use of the physical CPUs among the virtual machines.
- The VMM has a number of physical CPUs available and a number of threads to run on those CPUs. The threads can be VMM threads or guest threads. Guests are configured with a certain number of virtual CPUs at creation time, and that number can be adjusted throughout the life of the VM. When there are enough CPUs to allocate the requested number to each guest, the VMM can treat the CPUs as dedicated and schedule only a given guest's threads on that guest's CPUs.
- The VMM itself needs some CPU cycles for guest management and I/O management and can steal cycles from the guests
- Overcommitment, in which the guests are configured for more CPUs than exist in the system. Here, a VMM can use standard scheduling algorithms to make progress on each thread but can also add a fairness aspect to those algorithms.
- Even given a scheduler that provides fairness, any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will most likely be negatively affected by virtualization.
- The net outcome of such scheduling is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all of the cycles and indeed are scheduling all of the cycles.
- Commonly, the time-of-day clocks in virtual machines are incorrect because timers take longer to trigger than they would on dedicated CPUs. Virtualization can thus undo the scheduling-algorithm efforts of the operating systems within virtual machines. To correct for this, the VMM makes an application available for each type of operating system that the system administrator can install into the guests.

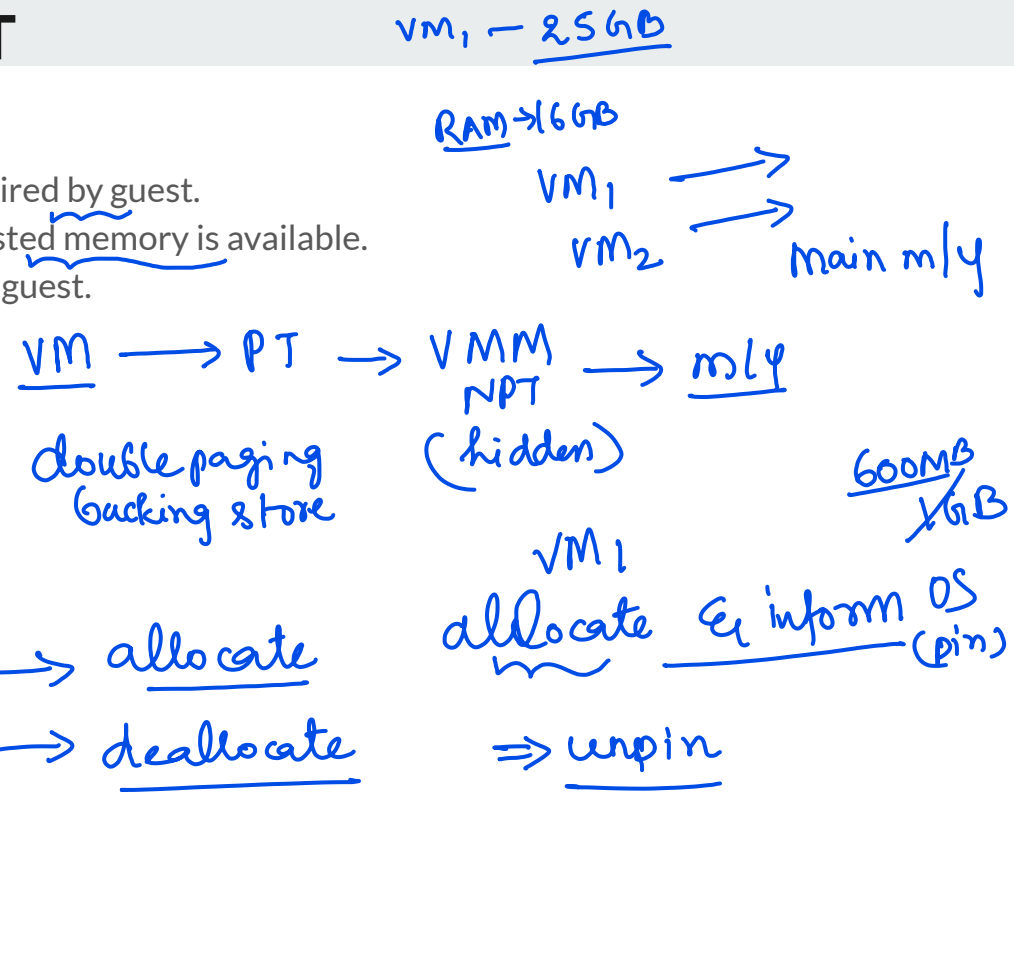
MEMORY MANAGEMENT

Work of VMM:

1. Calculates maximum memory size required by guest.
2. Provides illusion that always the requested memory is available.
3. Computes target real memory for each guest.

Recall Memory:

1. VM - PT , VMM -NPT
2. Pseudo device driver at each guest
3. Remove copies and hash



MEMORY MANAGEMENT

1. In virtualized environments, there are more users of memory (the guests and their applications, as well as the VMM), leading to more pressure on memory use.
2. VMMs typically overcommit memory, so that the total memory allocated to guests exceeds the amount that physically exists in the system.
3. VMM first evaluates each guest's maximum memory size.
4. VMMs must maintain the illusion that the guest has that amount of memory. Next, the VMM computes a target real-memory allocation for each guest based on the configured memory for that guest

HOW RECALLED:

1. Guest believes it controls memory allocation via its pagetable management, whereas in reality the VMM maintains a nested page table that translates the guest page table to the real page table.One approach is to provide double paging. Here, the VMM has its own page-replacement algorithms and loads pages into a backing store that the guest believes is physical memory.
2. Install in each guest a pseudo– device driver or kernel module that the VMM controls.
    a. (A pseudo–device driver uses device-driver interfaces, appearing to the kernel to be a device driver, but does not actually control a device. Rather, it is an easy way to add kernel-mode code without directly modifying the kernel.) This balloon memory manager communicates with the VMM and is told to allocate or deallocate memory. If told to allocate, it allocates memory and tells the operating system to pin the allocated pages into physical memory.
    b. To the guest, these pinned pages appear to decrease the amount of physical memory it has available, creating memory pressure. The guest then may free up other physical memory to be sure it has enough free memory.
    c. If memory pressure within the entire system decreases, the VMM will tell the balloon process within the guest to unpin and free some or all of the memory, allowing the guest more pages for its use.
3. Another common method for reducing memory pressure is for the VMM to determine if the same page has been loaded more than once. If this is the case, the VMM reduces the number of copies of the page to one and maps the other users of the page to that one copy. VMware, for example, randomly samples guest memory and creates a hash for each page sampled.

I/O

1. Specific devices to guest OS → I_1, I_2
2. Dedicated I/O
3. Idealized device drivers

VM_1 VM_2

VMM

H/W

I_1 I_{10}

Shared VMM

$VM_1 \rightarrow I_1$ ⇒ protection

VM_2

n/w

OS → CPU → I/P

VM_1 — VMM — I/O

I/P

① $VM_1 - IP_1$

$VM_2 - IP_2$

direct access

→ no hypervisor involvement

→ faster I/O

Eg. Type0

Type-2 H/W

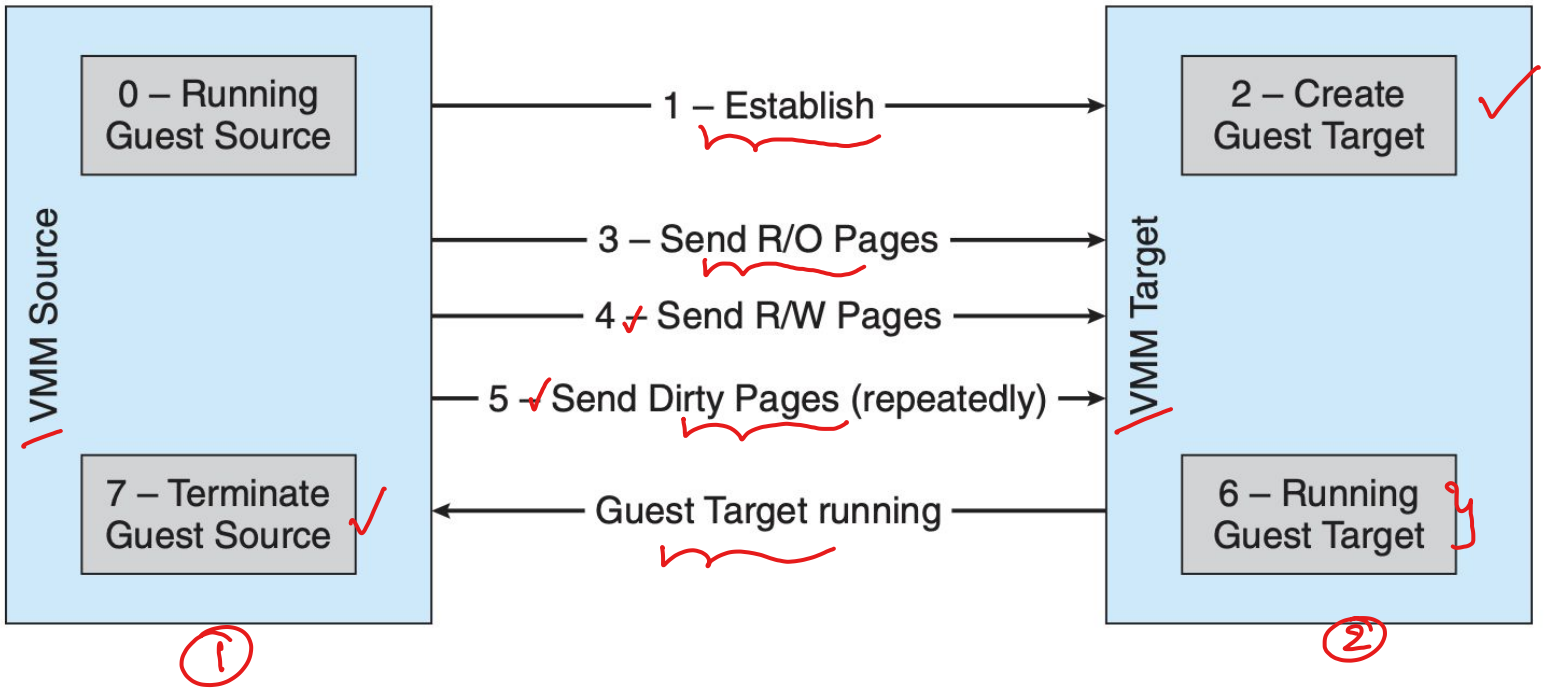
I/O MANAGEMENT

- I/O devices may be dedicated to guests, for example, or the VMM may have device drivers onto which it maps guest I/O. The VMM may also provide idealized device drivers to guests. In this case, the guest sees an easy-to-control device, but in reality that simple device driver communicates to the VMM, which sends the requests to a more complicated real device through a more complex real device driver. I
- The reason to allow direct access is to improve I/O performance. The less the hypervisor has to do to enable I/O for its guests, the faster the I/O can occur. With type 0 hypervisors that provide direct device access, guests can often run at the same speed as native operating systems. When type 0 hypervisors instead provide shared devices, performance may suffer.
- With direct device access in type 1 and 2 hypervisors, performance can be similar to that of native operating systems if certain hardware support is present.
- In addition to direct access, VMMs provide shared access to devices. Consider a disk drive to which multiple guests have access. The VMM must provide protection while the device is being shared, assuring that a guest can access only the blocks specified in the guest's configuration. In such instances, the VMM must be part of every I/O, checking it for correctness as well as routing the data to and from the appropriate devices and guests.
- each guest needs at least one IP address, because that is the guest's main mode of communication. Therefore, a server running a VMM may have dozens of addresses, and the VMM acts as a virtual switch to route the network packets to the addressed guests. The guests can be "directly" connected to the network by an IP address that is seen by the broader network (this is known as bridging). Alternatively, the VMM can provide a network address translation (NAT) address.

STORAGE

- **Type 0 hypervisors** often allow root disk partitioning, partly because these systems tend to run fewer guests than other systems. Alternatively, a disk manager may be part of the control partition, and that disk manager may provide disk space (including boot disks) to the other partitions.
- **Type 1 hypervisors** store the guest root disk (and configuration information) in one or more files in the file systems provided by the VMM. Type 2 hypervisors store the same information in the host operating system's file systems. In essence, a disk image, containing all of the contents of the root disk of each of the guests, is contained in one file in the VMM.
- If the administrator wants a **duplicate of the guest** (for testing, for example), she simply copies the associated disk image of the guest and tells the VMM about the new copy. Booting the new virtual machine brings up an identical guest. Moving a virtual machine from one system to another that runs the same VMM is as simple as halting the guest, copying the image to the other system, and starting the guest there.
- Frequently, VMMs provide a mechanism to **capture a physical system** as it is currently configured and convert it to a guest that the VMM can manage and run. This physical-to-virtual (P-to-V) conversion reads the disk blocks of the physical system's disks and stores them in files on the VMM's system or on shared storage that the VMM can access. VMMs also provide a virtual-tophysical (V-to-P) procedure for converting a guest to a physical system.

Live Migration of Guest Between Servers



1. The source VMM establishes a connection with the target VMM and confirms that it is allowed to send a guest.
2. The target creates a new guest by creating a new VCPU, new nested page table, and other state storage.
3. The source sends all read-only memory pages to the target.
4. The source sends all read–write pages to the target, marking them as clean.
5. The source repeats step 4, because during that step some pages were probably modified by the guest and are now dirty. These pages need to be sent again and marked again as clean.
6. When the cycle of steps 4 and 5 becomes very short, the source VMM freezes the guest, sends the VCPU's final state, other state details, and the final dirty pages, and tells the target to start running the guest. Once the target acknowledges that the guest is running, the source terminates the guest.