Virtual Machines

COMP3301/COMP7308 Operating Systems

Virtual Machines

- Overview
- History
- Benefits and Features
- Building Blocks
- Types of Virtual Machines and Their Implementations
- Virtualisation and Operating-System Components
- Examples

Chapter Objectives

- To explore the history and benefits of virtual machines
- To discuss the various virtual machine technologies
- To describe the methods used to implement virtualisation
- To show the most common hardware features that support virtualisation and explain how they are used by operating-system modules

Overview

- Fundamental idea - abstract hardware of a single computer into several different execution environments
  - Similar to layered approach
  - But layer creates a virtual system, or **virtual machine** (**VM**) on which operating systems can run

What is a Computer?

- A computer is a CPU and memory
  - Memory is accessed by the CPU over a bus
  - The CPU loads instructions from memory
  - CPU instructions load memory into registers, store registers to memory, operate on registers, checks register values, and jumps to new instruction locations
- Instructions are a series of bytes or words
  - You can read instructions as data and reason about their effects

CPU Instructions

```
$ objdump -dl test | sed -n -e '/^main():/,/^$/ p'
main():
    13a0: 55                      push   %rbp
    13a1: 48 89 e5                mov    %rsp,%rbp
    13a4: 48 83 ec 20             sub    $0x20,%rsp
    13a8: c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffffffffffc(%rbp)
    13af: 89 7d f8                mov    %edi,0xfffffffffffffff8(%rbp)
    13b2: 48 89 75 f0             mov    %rsi,0xfffffffffffffff0(%rbp)
    13b6: 8b 75 f8                mov    0xfffffffffffffff8(%rbp),%esi
    13b9: 48 8d 3d 10 f1 ff ff    lea    -3824(%rip),%rdi # 4d0 <__EH_FRAME_BEGIN__-0x40>
    13c0: b0 00                   mov    $0x0,%al
    13c2: e8 e9 00 00 00          callq  14b0 <printf@plt+0x60>
    13c7: 31 f6                   xor    %esi,%esi
    13c9: 89 45 ec                mov    %eax,0xffffffffffffffec(%rbp)
    13cc: 89 f0                   mov    %esi,%eax
    13ce: 48 83 c4 20             add    $0x20,%rsp
    13d2: 5d                      pop    %rbp
    13d3: c3                      retq
```

CPU Emulation

- A program can model or emulate a CPU and memory
  - A logical or virtual CPU is represented by a data structure containing registers and their state
  - Memory can be emulated as memory with bounds checks
  - Load code into the virtual memory, point the instruction pointer at it, and go
  - Parse the loaded instructions and apply their effects to the CPU data structure and virtual memory state

What is a Computer?

- A computer has peripherals
  - eg, serial ports, disk controllers, network interfaces, etc
  - "Host" devices sit on the CPU/memory bus
- A CPU interacts with these devices via memory operations
  - device registers (a register window) is accessed at a memory location via the bus
  - a CPU uses a memory load operation to read a device register, a store operation to write a device register

Devices

- Devices have state that is updated by register accesses, or actual hardware change
  - register reads and writes can have different effects, eg, serial ports have a single register used for both transmit and receive
- A CPU can poll a device for a hardware change, or wait for an interrupt
  - an interrupt avoids busy waiting or wasting time on polling
  - effectively causes the CPU to jump to special "interrupt service routine" code

Device Emulation

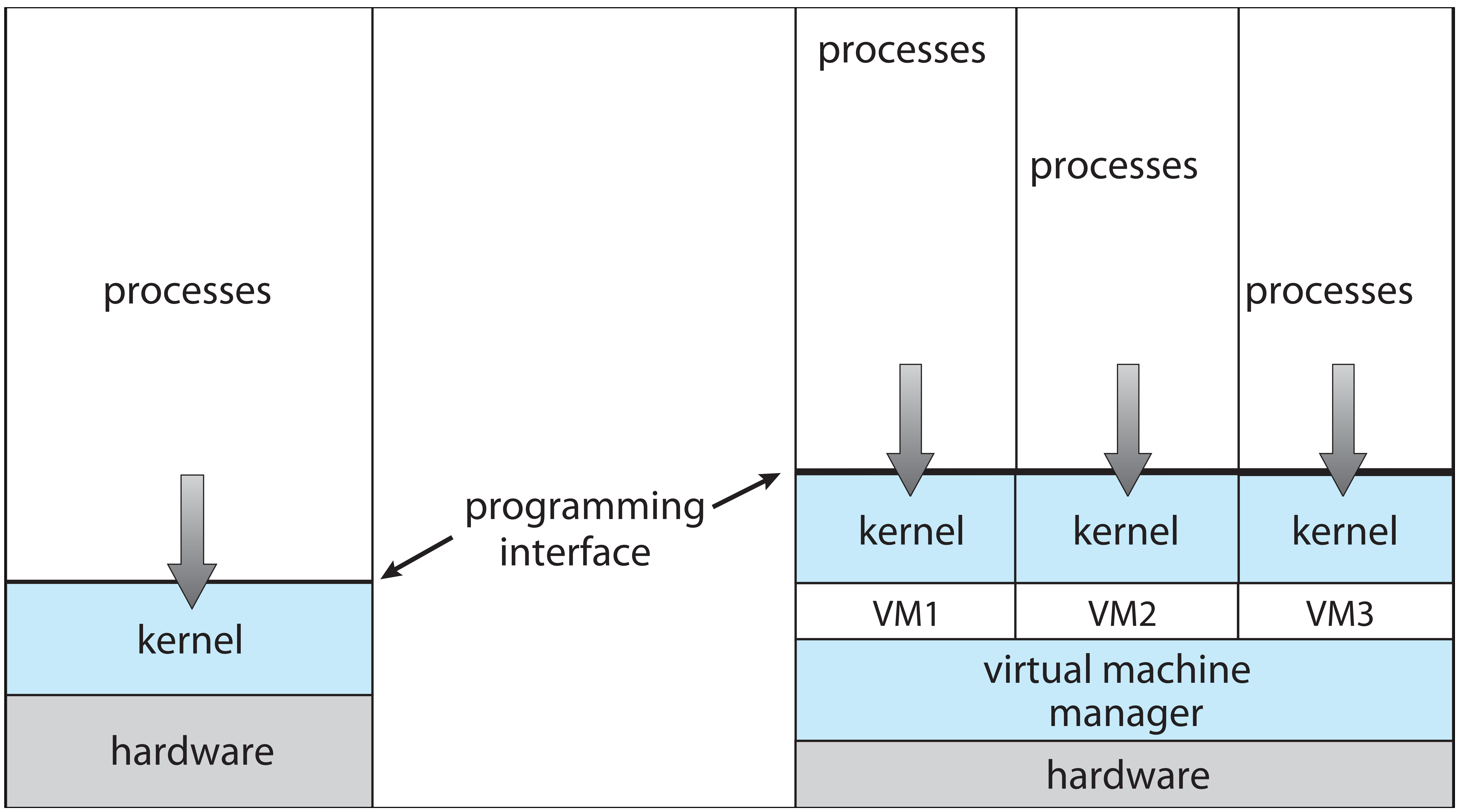
- Like a CPU, devices have state, which you can reason about changes to
  - device state is represented as a data structure too
- The CPU virtual memory emulation can special case register windows
  - register reads and writes become "messages" for the device emulation to process
  - "store" messages contain the address (register location) and value
  - "read" messages contain the address (register location) and produce a value to place in a CPU register

Emulation

- Pro: any guest CPU can be emulated on any host CPU
  - eg, qemu can run arm code on x86 CPUs
- Con: very slow
  - guest CPU instruction execution is orders of magnitude slower
- Solution: let code run natively on the CPU.
  - extend the operating system process facility to provide a virtual machine interface

VM Components

- **Host** - underlying hardware system
- **Virtual machine manager** (VMM) or **hypervisor** - create and runs virtual machines by providing an interface that is equivalent to a physical host
    - Except in the case of paravirtualisation
- **Guest** - a process providing a virtual implementation of a physical host
    - Usually an operating system
- A single host can run multiple operating systems concurrently, each in a VM



Non-virtual machine

Virtual machine

History

- The textbook says virtual machines first appeared commercially on IBM mainframes in 1972 in the IBM VM/370 operating system on System/370 mainframes
  - VM-CP allowed multiple instances of guest operating systems to run on a single mainframe
  - Guest operating systems included CMS, or even VM-CP itself
  - Still exists today, and now supports running AIX and Linux
- Could be argued that the real origin of virtual machines was in the 1960s with the Honeywell 200 series machines running IBM 1401 software

History

- Formal definition of virtualisation helped move it beyond IBM

1. A VMM provides an environment for programs that is essentially identical to the original machine
2. Programs running within that environment show only minor performance decreases
3. The VMM is in complete control of system resources

Implementation of VMMs

- Type 0 hypervisors - Hardware-based solutions that provide support for virtual machine creation and management via firmware
    - eg, IBM LPARs and Oracle LDOMs
- Type 1 hypervisors - Operating system like software built to provide virtualisation
    - eg, VMware ESXi, Citrix/Xen Server
- Type 2 hypervisors - A general purpose operating systems that provides standard functionality as well as VMM functionality
    - eg, KVM on Linux, bhyve on FreeBSD, Hypervisor.framework on macOS

Implementation of VMMs

- Other variants include:
  - Paravirtualisation - A technique where the guest OS is modified to work in cooperation with the VMM to optimise performance.
  - Emulators - Interpret the execution of a machine completely in software
    - eg, DOSBox, qemu (without KVM), SimH, gxemul
  - Containers - virtualisation or isolation of a kernel interface instead of a host
    - eg, Solaris Zones, BSD Jails, AIX WPARs, Docker

Types of Virtual Machines

- Many variations as well as hardware details
  - Assume VMMs take advantage of hardware features
  - HW features can simplify implementation, improve performance
- Devices are generally still emulated
  - Guest access to physical devices is possible
  - VMM specific devices exist to mitigate overhead, or provide special functionality

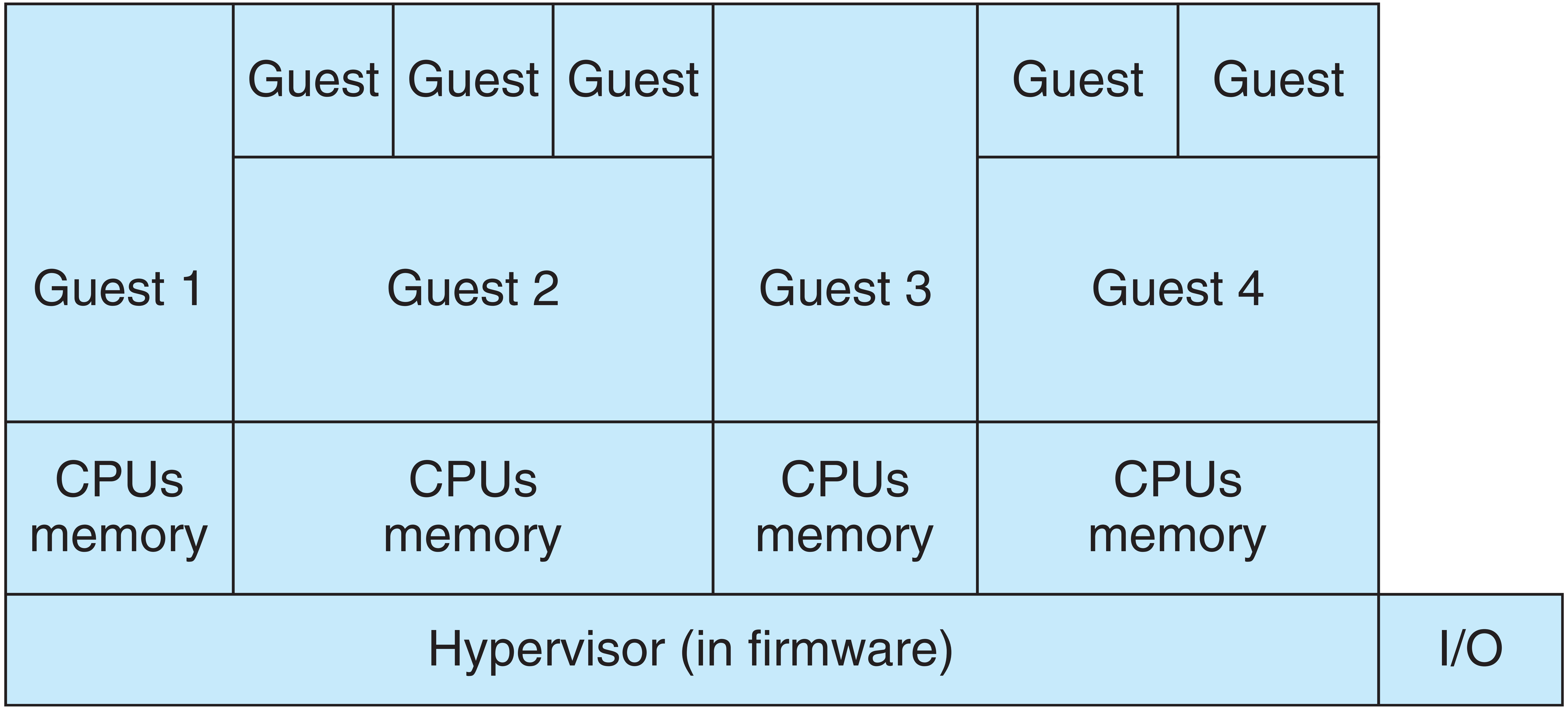
Types of Virtual Machines

- Whatever the type, a VM has a lifecycle
  - Created by VMM
  - Resources assigned to it (number of cores, amount of memory, networking details, storage details)
    - In Type 0 hypervisor, resources usually dedicated
  - Other types dedicate or share resources, or a mix
  - When no longer needed, VM can be deleted, freeing resources

Type 0 Hypervisor

- Old idea, under many names by HW manufacturers
- A "platform" feature implemented by firmware
  - OS does not need to do anything special, VMM is in firmware
  - Smaller feature set than other types
  - Each guest generally 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

Type 0 Hypervisor



Type 1 Hypervisor

- Special purpose operating systems that run natively on HW
  - Rather than providing system call interface, create run, and manage guest OSes
- Implement device drivers for host HW because no other component can
  - However, may use a special VM to support the physical hardware and provide services to other VMs
- Also provide other traditional OS services like CPU and memory management

Type 1 Hypervisor

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

Type 2 Hypervisor

- A general purpose OS that also provides VMM functionality
  - eg, Linux with KVM, Windows with Hyper-V
- Perform normal duties as well as VMM duties
- In many ways, treat guests OSes as just another process
  - Albeit with special handling when guest tries to execute special instructions
- Reuses existing driver functionality in the host OS/VMM to support hardware

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, like a Process Control Block (PCB) for normal programs

Trap and Emulate

- Dual mode CPU means guest VM executes in user mode
  - Not safe to let guest kernel run in kernel mode too
  - So VM needs two extra 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

- 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, analyses error, emulates operation as attempted by guest
  - Returns control to guest in virtual kernel mode, but actually in user mode
- Known as trap-and-emulate
  - Most virtualisation products use this at least in part

Trap and Emulate

- 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 virtualisation performance

Binary Translation

- Some CPUs don’t have clean separation between privileged and non-privileged instructions
- 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, but no trap is generated

Binary Translation

- Other similar problem instructions we will call special instructions
  - Trap-and-emulate method considered impossible until 1998 and the development of binary translation

1. If guest VCPU is in user mode, guest can run instructions natively
2. If guest VCPU in virtual kernel mode (guest believes it is in kernel mode):
   - VMM parses instructions and rewrites them
   - Non-special-instructions run natively
   - Special instructions translated to equivalent code that perform

Binary Translation

- Performance of this method would be poor without optimisations
  - 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 a guest believing it controls the page tables, and the VMM that actually controls the tables?
- Common method (for trap-and-emulate and binary translation) is nested page tables (NPTs)
  - Each guest OS maintains page tables to translate virtual to physical addresses
  - VMM maintains per guest NPTs to represent guest’s page-table state
  - When a guest tries to change page table, the VMM updates NPTs
  - When guest runs, the VMM updates the system page tables based on the NPTs

Hardware Assistance

- There are obvious deficiencies with the initial building blocks on x86
- Intel added new VT-x instructions in 2005, and AMD the AMD-V instructions in 2006
  - Mitigates the need for binary translation
  - Generally defines 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)

Hardware Assistance

- Still traps to VMM on access to virtualised devices and privileged instructions, but far less
- Assistance improves over time
  - HW support for Nested Page Tables, DMA, interrupts as well over time
  - A VMM using modern features is a lot less work to implement

Virtualisation vs the OS

- Now let's look at operating system aspects of virtualisation
  - 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?

CPU Scheduling

- Even single-CPU systems can act like multiprocessor ones when virtualised
  - One or more virtual CPUs per guest
- Generally VMM has one or more physical CPUs and vCPUs run as threads on them
- Guests configured with certain number of vCPUs
  - Can be adjusted throughout life of VM, CPU hot-plug is cheap in software

CPU Scheduling

- When enough CPUs for all guests -> VMM can allocate dedicated CPUs, each guest much like native operating system managing its CPUs
  - Usually not enough CPUs -> CPU overcommitment
- VMM can use standard scheduling algorithms to put threads on CPUs
  - Some add fairness aspect
  - But VMM has very little visibility of workload inside a guest

CPU Scheduling

- Cycle stealing by VMM and oversubscription of CPUs means guests don’t get CPU cycles they expect
  - Consider timesharing scheduler in a guest trying to schedule 100ms time slices -> each may take 100ms, 1 second, or longer
  - Poor response times for users of guest
  - Time-of-day clocks drift
- Some VMMs provide applications or guest devices to fix time-of-day and provide other integration features

Memory Management

- A computer boots with a fixed amount of RAM, and thinks it owns it
  - VMs therefore get allocated all their memory up front and own it for their lifetime
  - Traditional programs allocate memory on demand and release it when it's unused
- Memory is generally the most contended resource in a VMM
  - ie, it is the limiting factor on the number of concurrent VMs a host may run

Memory Management

- VMMs may mitigate guest memory usage by
  - Double-paging, or swapping "idle" guest pages to a backing store like disk or SSD
  - Have the guest use a "memory balloon" driver that "allocates" guest physical memory, and advertises these allocations to the VMM for it to use
  - Deduplicate memory pages between guests
    - Multiple virtual instances of the same OS could share their kernel image in host physical memory

I/O

- Guests want to store data on disk, or communicate with a network
- VMM must emulate existing hardware to provide I/O, or provide optimised virtual hardware and drivers for guests
- But overall I/O is complicated for VMMs
  - The less hypervisor needs to do for I/O for guests, the better
  - Possibilities include direct device access, DMA pass-through, direct interrupt delivery
    - Again, HW support needed for these

I/O

- Network access from the guest to the real world implemented via devices like tap(4)
  - Guest transmits a packet by posting to the emulated network interface device
  - VMM traps, collects the packet, and writes it to the host network stack for handling
- Storage can be similarly translated to reads and writes of a "backing device" in the VMM
  - eg, a file or an actual disk can support read/write operations on behalf of the guest

Live Migration

- VMMs can offer features unavailable on actual hardware, eg, migrating a virtual machine between physical hosts
  - Running guest can be moved between systems, without interrupting user access to the guest or its apps

Live Migration

1. Source VMM connects to the VMM

2. The target VMM creates a new guest by creating vCPUs, devices, etc, etc

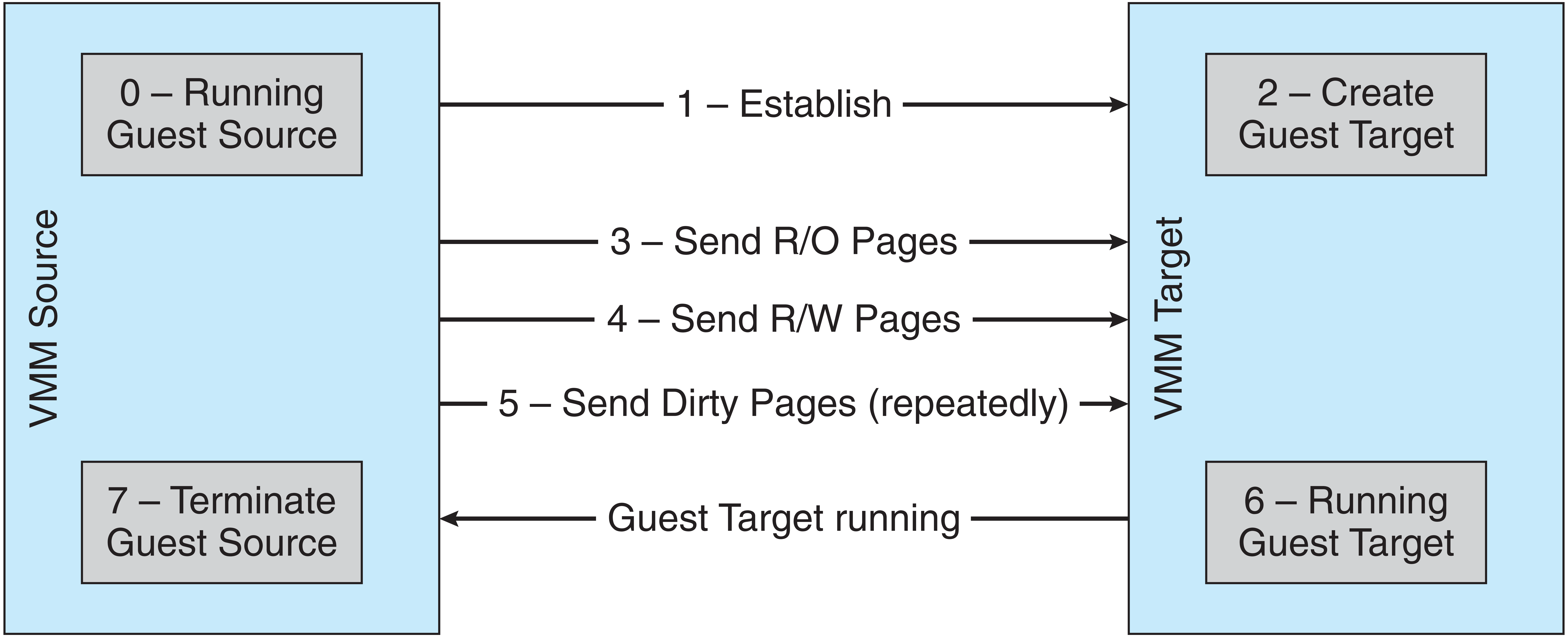
3. The source sends all read-only guest memory pages to the target

4. The source sends all read-write pages to the target, marking them as clean

5. The source repeats step 4, as during that step some pages were probably modified by the guest and are now dirty

6. When cycle of steps 4 and 5 becomes very short, source VMM freezes guest, sends VCPU’s final state, sends other state details, sends final dirty pages, and tells target to start running the guest

7. Once target acknowledges that guest running, source terminates guest

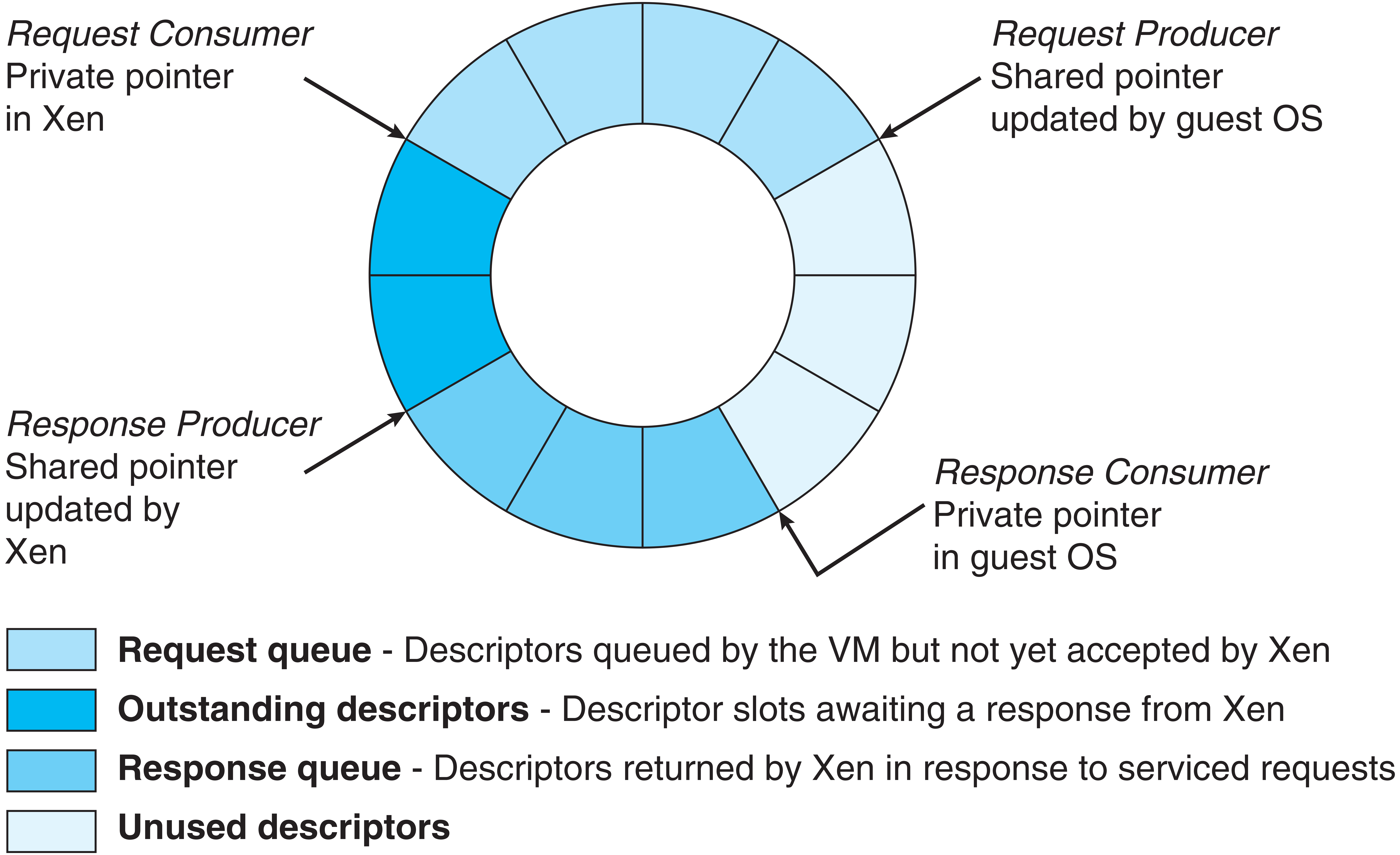

0 – Running Guest Source
VMM Source
7 – Terminate Guest Source
1 – Establish
3 – Send R/O Pages
4 – Send R/W Pages
5 – Send Dirty Pages (repeatedly)
Guest Target running
2 – Create Guest Target
VMM Target
6 – Running Guest Target

Paravirtualisation

- Does not fit the definition of virtualisation – 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

Paravirtualisation

- Xen, leader in paravirtualised 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


Request Consumer
Private pointer
in Xen
Request Producer
Shared pointer
updated by guest OS
Response Producer
Shared pointer
updated by
Xen
Response Consumer
Private pointer
in guest OS
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

Paravirtualisation

- Xen memory management does not include nested page tables
    - Each guest has read-only page tables
    - Guest uses hypercall (call to hypervisor) when page-table changes are needed
    - HW assistance for VM extended page tables means this is unnecessary on modern systems

Containers

- Some goals of virtualisation are segregation of apps, performance and resource management, easy start, stop, move, and management of them
  - Can do those things without full-fledged hardware virtualisation
  - If applications are already compiled for the host operating system, don't need full virtualisation to meet these goals
  - Containers use virtualisation of kernel resources, not hardware resources

Solaris (Illumos) Zones

- Only one kernel running - the host OS
- The kernel adds a "scope" to historically global kernel resources
  - eg, process tables, network interfaces and stack, filesystem visibility, devices, etc
- The kernel has full visibility of processes and their resources
  - Able to make better scheduling decisions across all scopes
- Processes have high fidelity access to resources like time and memory

LX Brand Zones

- Solaris implemented Linux syscall emulation
  - Illumos picked it up and polished it
  - Allows Linux binaries to run natively on the Solaris Kernel
- Combined with zones allows for a branded zone
  - Feels like a native Linux system

Docker

- Linux introduced "namespaces" that allow partitioning of specific resources
    - eg, Process namespace, network namespace, filesystem namespace
- Docker combined these to produce what looks like an isolated runtime environment
- Also provided a tool to manage the lifecycle and production of these environments
    - A lot easier to get started with than zones on Solaris

Joyent Triton

- A cloud orchestration suite
- Runs an Illumos fork called SmartOS on a collection of "compute nodes" or hosts
- Supports provisioning of containers on and across a cluster of compute nodes
- Developed Docker compatibility, allowing Linux based Docker containers to be deployed across a fleet of SmartOS machines

Benefits and Features

- Operational benefits
  - Operating system and software lifecycles/licensing can be managed independently of the physical host
  - VMMs can implement features to support administration including snapshotting and restore points, live migration, templating
  - Workload consolidation, eg, development, testing, and production can use the same hardware
  - Automation of OS provisioning, and configuration (ultimately cloud computing)

Benefits and Features

- Isolation - host system protected from VMs, VMs protected from each other
    - conditions apply
- Great for OS research and development

Caveats

- Virtualisation, while cheaper than emulation, is still not free
  - Low latency systems or applications may still justify "bare metal" deployments
- Correct operation of the VMM is crucial for providing isolation
  - Particularly in multi tenant environments
  - A VMM requires correct operation of the host platform

return;