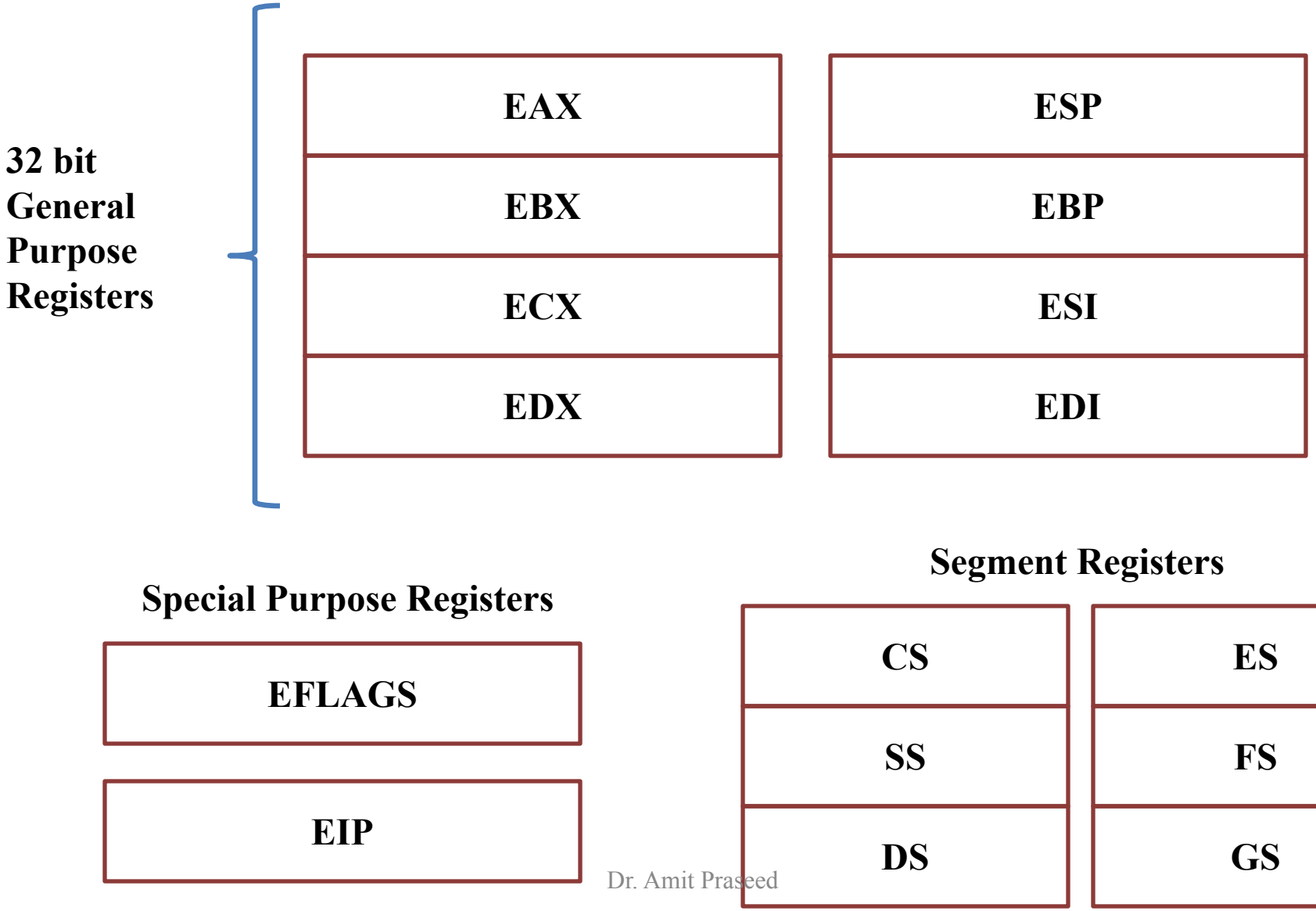
A Review of x86 Architecture

Dr. Amit Praseed

Overview of x86 Architecture

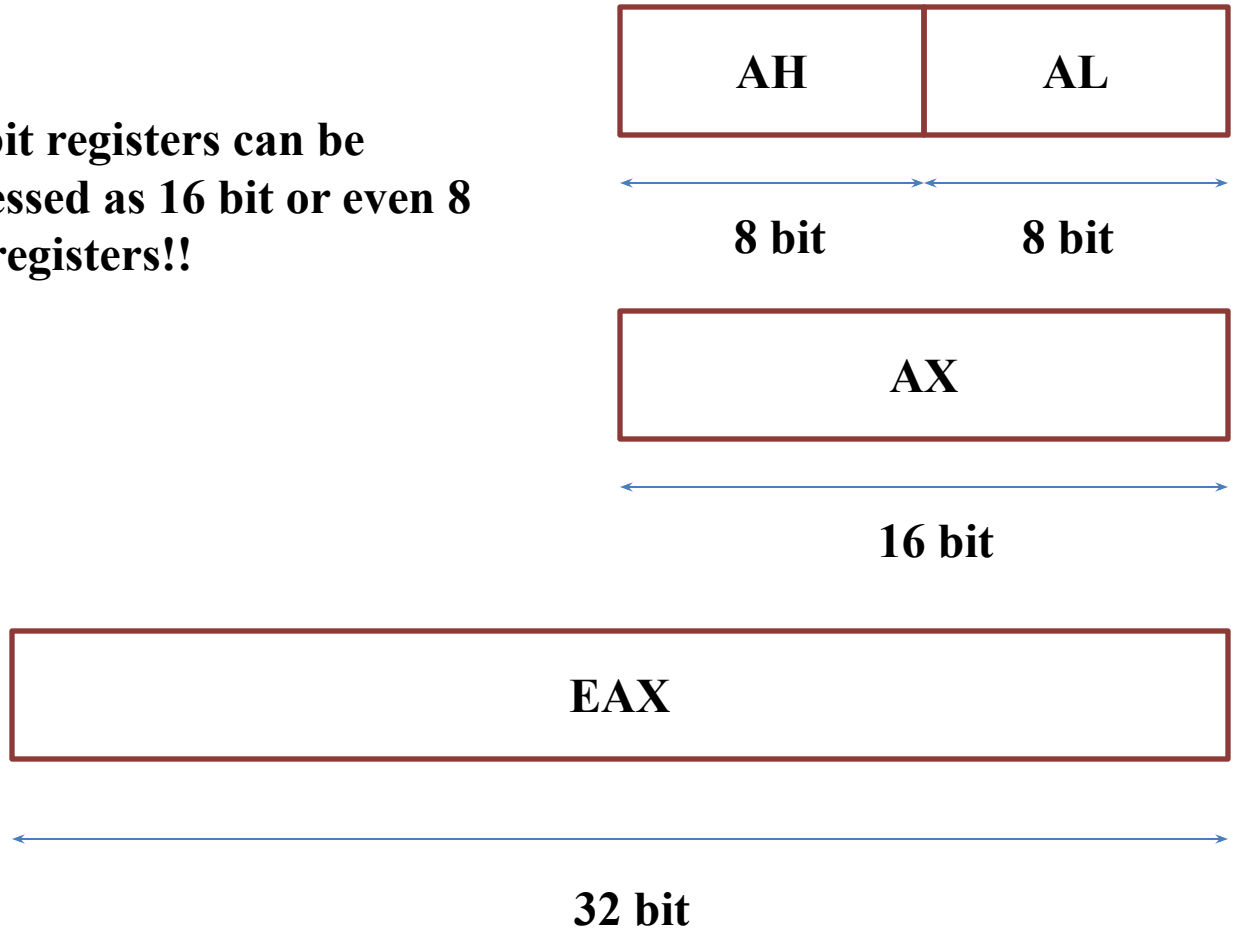
- **CISC Architecture**
  - Complex Instruction Set Computing
- **4 GB addressable memory**
  - 32 bit address space

Registers


32 bit General Purpose Registers
EAX
EBX
ECX
EDX
ESP
EBP
ESI
EDI
Special Purpose Registers
EFLAGS
EIP
Segment Registers
CS
SS
DS
ES
FS
GS

Registers

32 bit registers can be accessed as 16 bit or even 8 bit registers!!



Uses of Registers

- General-Purpose Registers
  - EAX – accumulator
  - ECX – loop counter
  - ESP – stack pointer (should never be used for arithmetic or data transfer)
  - ESI, EDI – index registers (used for high-speed memory transfer instructions)
  - EBP – extended frame pointer for stack

Uses of Registers

- Segment Registers
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES, FS, GS - additional segments
- EIP – instruction pointer
- EFLAGS
  - status and control flags
  - each flag is a single binary bit (set or clear)
- Some other system registers such as IDTR, GDTR LDTR etc.

Control Registers

- 8 control registers CR0 – CR7
  - CR0 : Logical processor functions, such as enabling/disabling protected mode or paging
  - CR2 : When a virtual to physical address conversion fails, the VA is latched into CR2 and an exception is raised
  - CR3 : Holds the memory address of the top level address translation table
  - CR4 : Used to control operations like virtual 8086 support, I/O breakpoints etc.
  - CR1, CR5-CR7 : Reserved

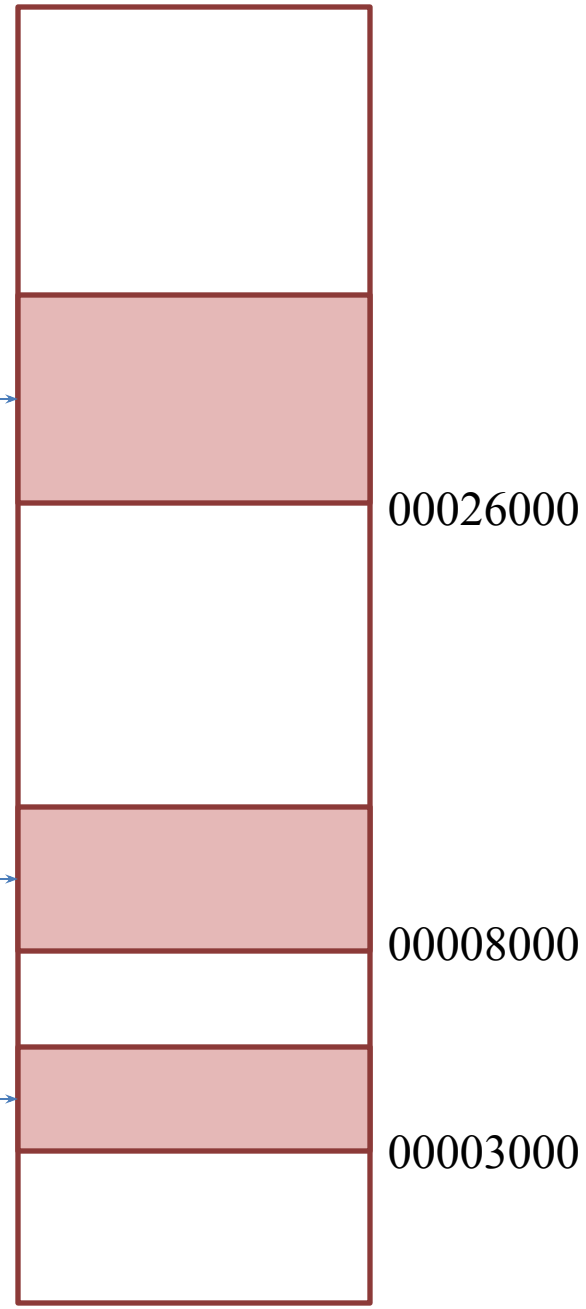
Protected Addressing Mode

- 4 GB addressable RAM (32-bit address)
  - (00000000 to FFFFFFFFh)
  - Each program assigned a memory partition which is protected from other programs
  - Designed for multitasking
  - Supported by Linux & MS-Windows
- Segment descriptor tables
- Program structure
  - code, data, and stack areas
  - CS, DS, SS segment descriptors
  - global descriptor table (GDT)

Segmentation in x86 Systems

- Every program contains of 3 logical segments – code segment (CS), data segment (DS) and stack segment (SS)
- These segments can be present anywhere in memory, but in continuous memory locations
- Local Descriptor Table (LBT) holds the base address and limit of these segments

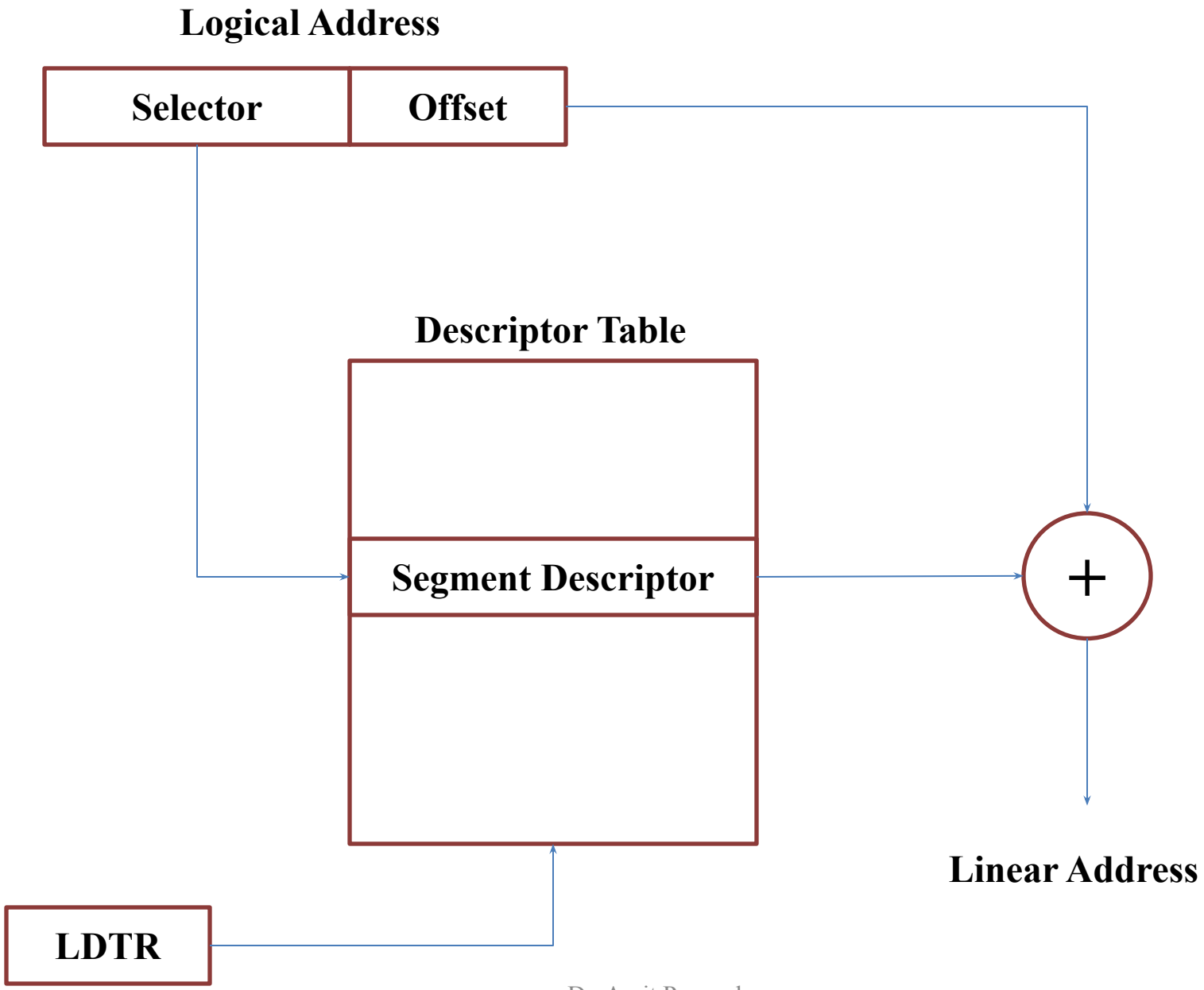
Base	Limit	Access
00026000	0010	
00008000	000A	
00003000	0002	



Limit is multiplied by 1000h

Address Translation in Segmentation

- The program refers to instructions using a base address and an offset
  - "4^{th} byte in the Data Segment"
- So address translation is relatively simple
  - Find out where the segment begins (refer the LDT)
  - Add the offset

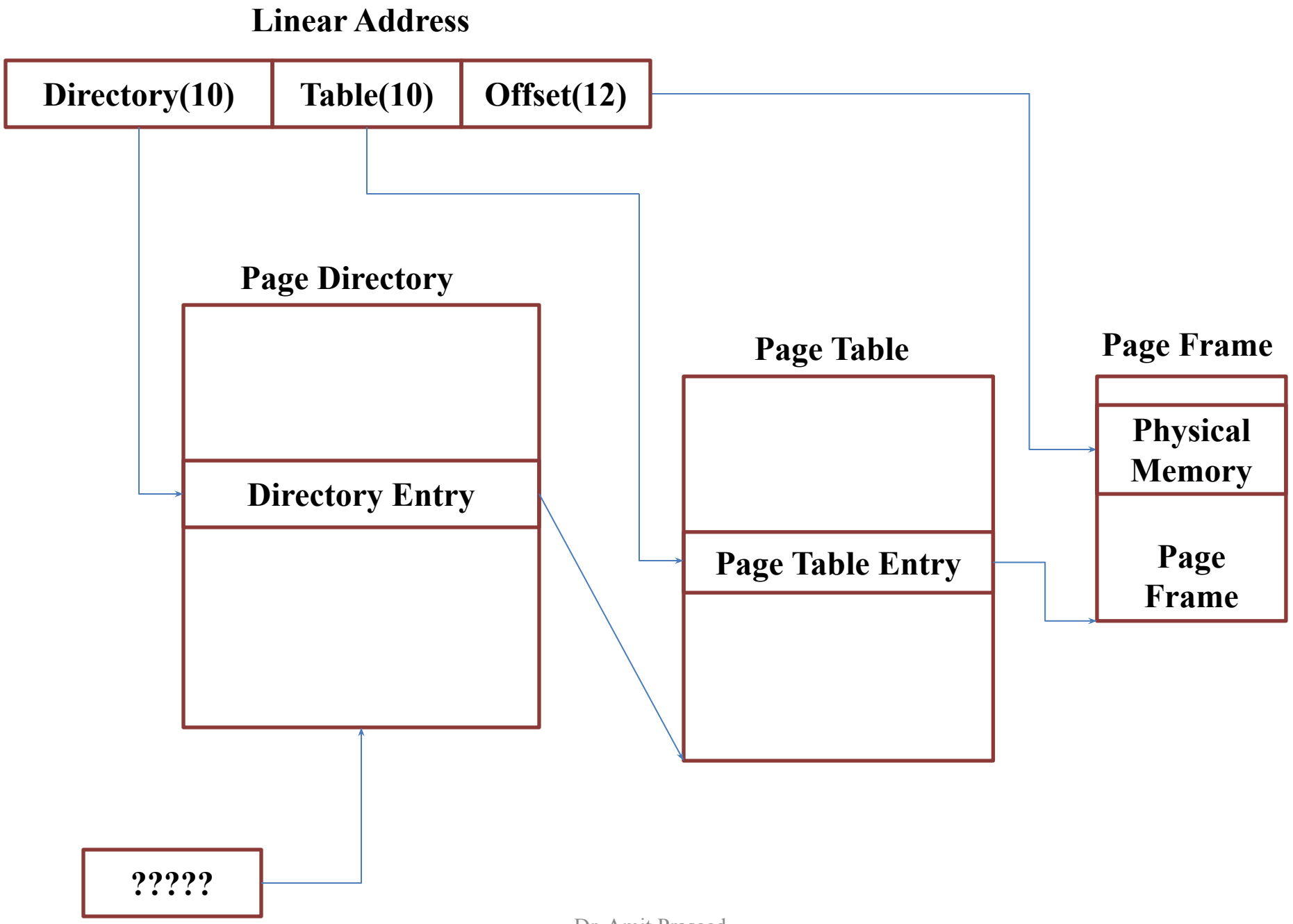

Logical Address
Selector
Offset
Descriptor Table
Segment Descriptor
+
Linear Address
LDTR

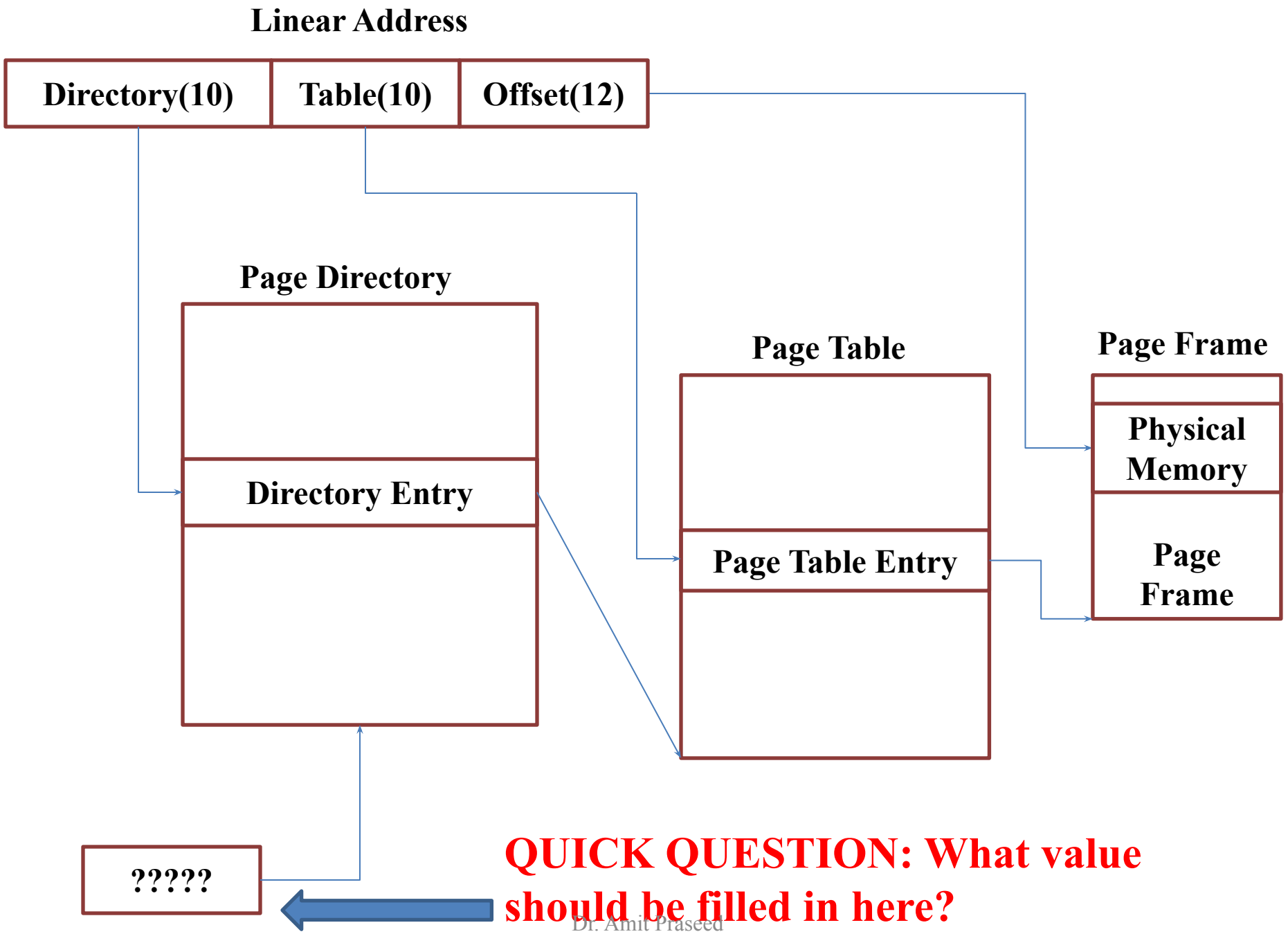
Paging in x86

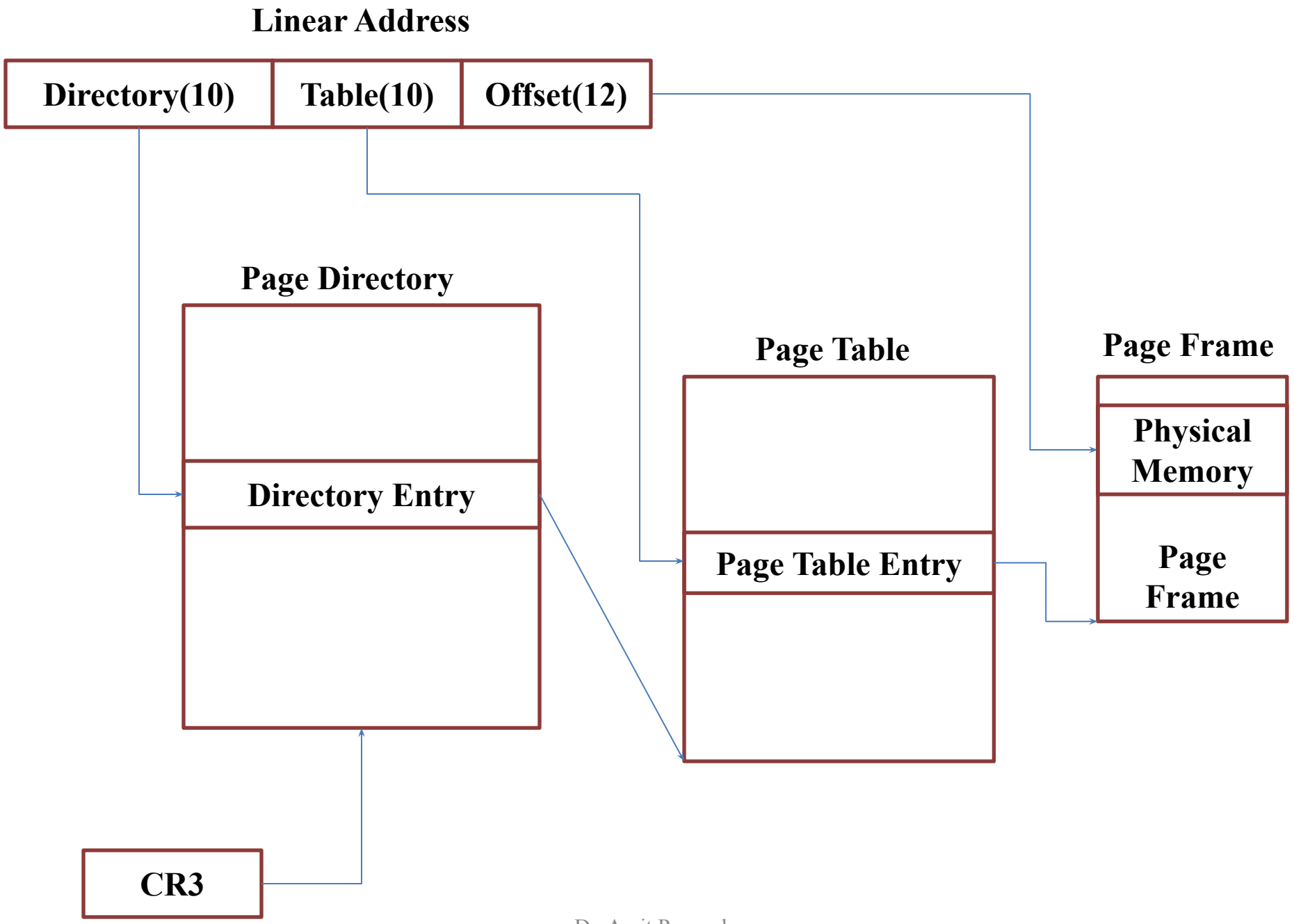
- Virtual memory uses disk as part of the memory, thus allowing sum of all programs can be larger than physical memory
- Only part of a program must be kept in memory, while the remaining parts are kept on disks
- The memory used by the program is divided into small units called pages (4096-byte)
- As the program runs, the processor selectively unloads inactive pages from memory and loads other pages that are immediately required.

Paging in x86

- OS maintains page directory and page tables
- Page translation: CPU converts the linear address into a physical address
- Page fault: occurs when a needed page is not in memory, and the CPU interrupts the program
- Virtual memory manager (VMM) – OS utility that manages the loading and unloading of pages
- OS copies the page into memory, program resumes execution


Linear Address
Directory(10)
Table(10)
Offset(12)
Page Directory
Directory Entry
Page Table
Page Table Entry
Page Frame
Physical Memory
Page Frame
?????


Linear Address
Directory(10)
Table(10)
Offset(12)
Page Directory
Directory Entry
Page Table
Page Table Entry
Page Frame
Physical Memory
Page Frame
?????
QUICK QUESTION: What value should be filled in here?

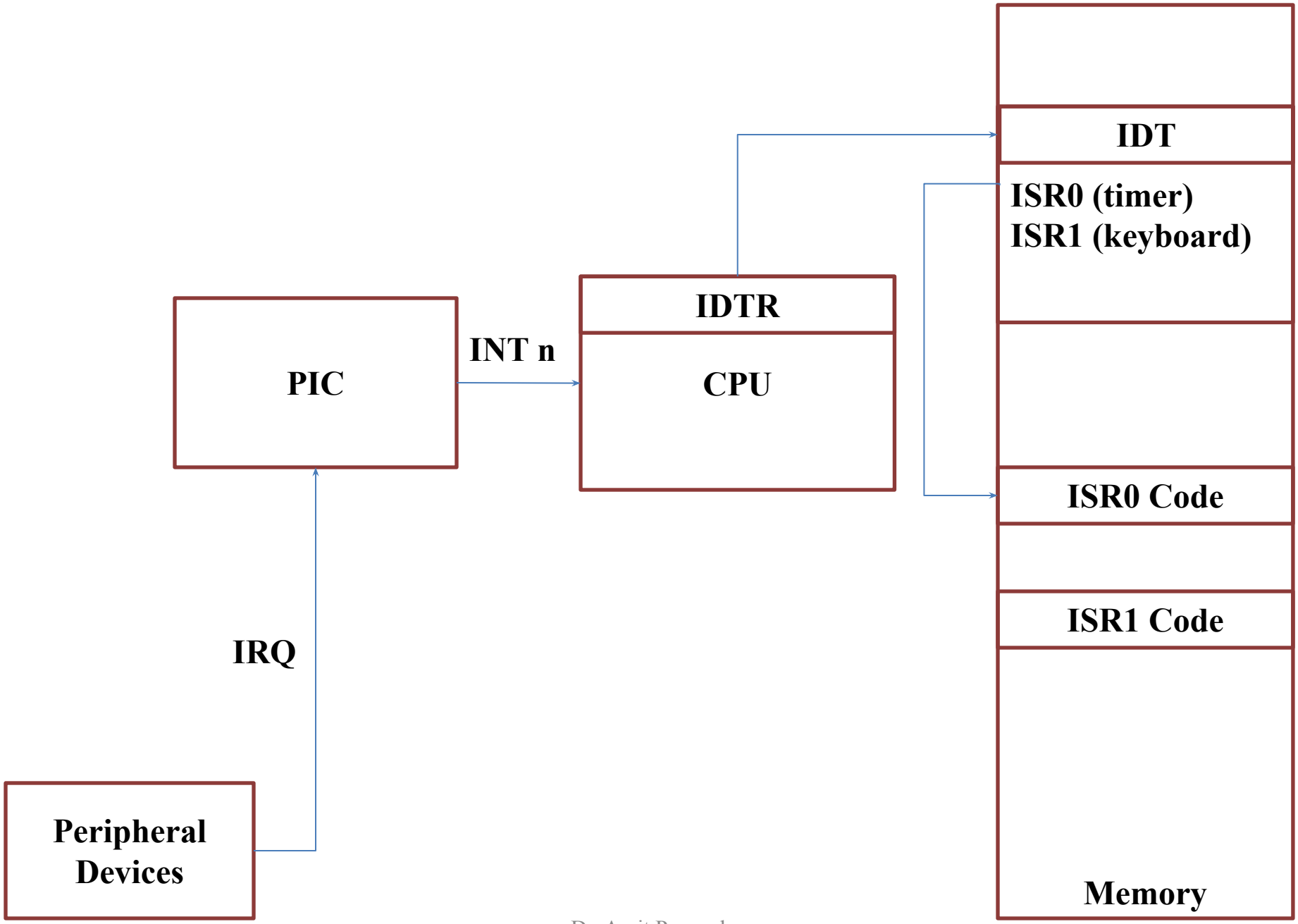

Linear Address
Directory(10)
Table(10)
Offset(12)
Page Directory
Directory Entry
Page Table
Page Table Entry
Page Frame
Physical Memory
Page Frame
CR3

Interrupts in x86

- Interrupts are events from devices to the CPU signalizing that device has something to tell, like user input on the keyboard or network packet arrival
  - Without interrupts, polling is the only option - introducing latency and being a horrible person.
- There are 3 sources or types of interrupts:
  - Hardware interrupts - comes from hardware devices like keyboard or network card.
  - Software interrupts - generated by the software int instruction. Before introducing SYSENTER/SYSEXIT system calls invocation was implemented via the software interrupt int $0x80.
  - Exceptions - generated by CPU itself in response to some error like “divide by zero” or “page fault”.

Interrupts in x86

- x86 interrupt system involves 3 parts to work conjointly:
  - **Programmable Interrupt Controller (PIC)** must be configured to receive interrupt requests (IRQs) from devices and send them to CPU.
  - CPU must be configured to receive IRQs from PIC and invoke correct interrupt handler, via gate described in an **Interrupt Descriptor Table (IDT)**.
  - Operating system kernel must provide **Interrupt Service Routines (ISRs)** to handle interrupts and be ready to be preempted by an interrupt. It also must configure both PIC and CPU to enable interrupts.

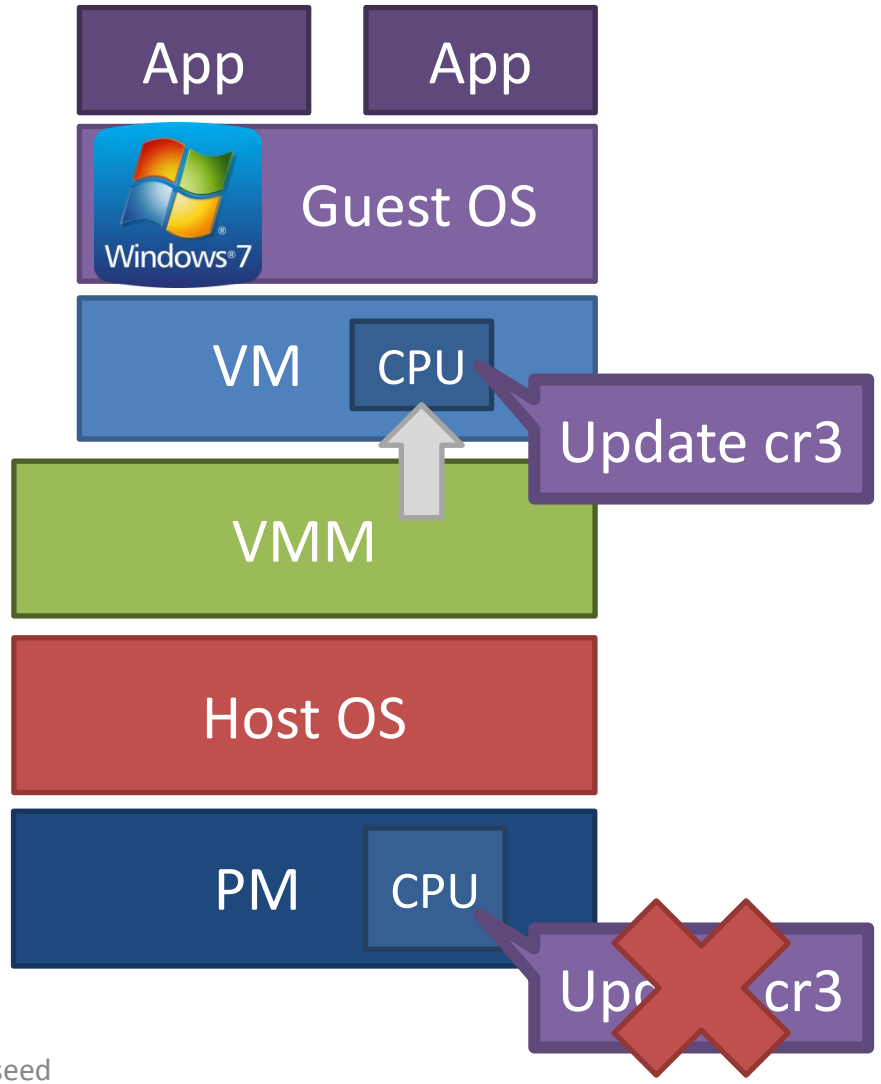

IDT
ISR0 (timer)
ISR1 (keyboard)
IDTR
CPU
PIC
INT n
ISR0 Code
ISR1 Code
IRQ
Peripheral Devices
Memory

X86 CPU Virtualization

Dr. Amit Praseed

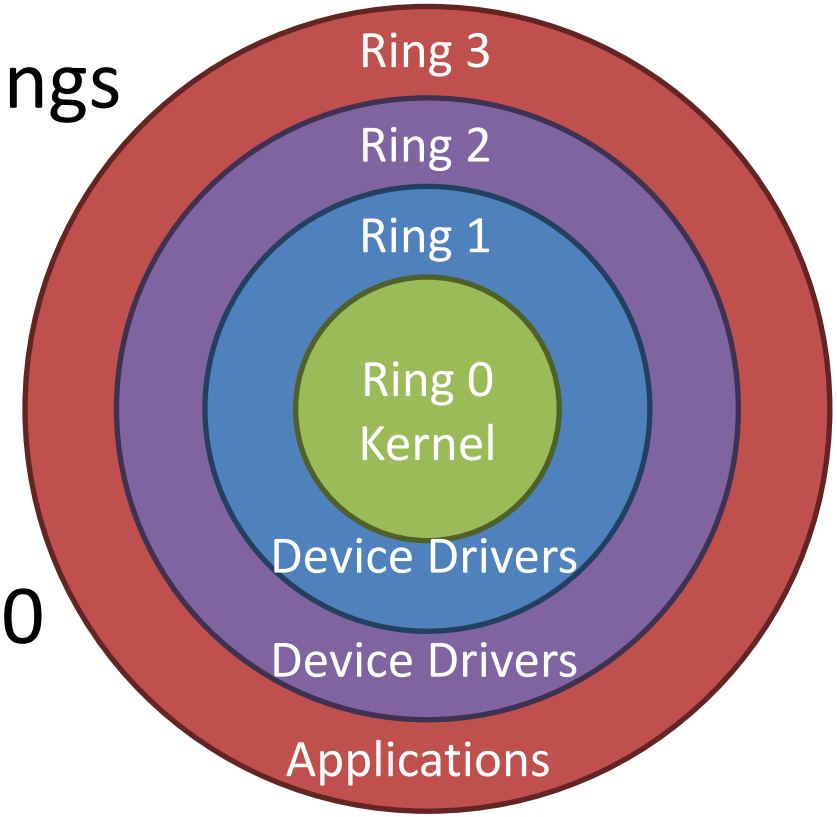
Virtual and Physical CPU

- Each guest has a virtual CPU created by the VMM
- However, the virtual CPU is only used to store state
  - E.g. if a guest updates *cr3* or *eflags*, the new value is stored in the virtual CPU
- Guest code executes on the physical CPU
  - Keeps guest performance high
  - Guests run in userland, so security is maintained



Protected Mode

- Most modern CPUs support protected mode
- x86 CPUs support three rings with different privileges
  - Ring 0: OS kernel
  - Ring 1, 2: device drivers
  - Ring 3: “userland”
- Most OSes only use rings 0 and 3



Privileged Instructions

- OSes rely on many privileges of ring 0
  - cri, sti, popf – Enable/disable interrupts
  - hlt – Halt the CPU until the next interrupt
  - mov cr3, 0x00FA546C – install a page table
  - Install interrupt and trap handlers
  - Etc…
- However, guests run in userland (ring 3)
- VMM must somehow virtualize privileged operations

Using Exceptions for Virtualization

- Ideally, when a guest executes a privileged instruction in ring 3, the CPU should generate an exception
- Example: suppose the guest executes hlt
  1. The CPU generates a protection exception
  2. The exception gets passed to the VMM
  3. The VMM can emulate the privileged instruction
     - If guest 1 runs hlt, then it wants to go to sleep
     - VMM can do guest1.yield(), then schedule guest 2

Types of Machine Instructions

- **Privileged instructions**
  - Executed in kernel mode
  - When attempted to be executed in user mode, they cause a trap and so executed in kernel mode.
- **Non-privileged instructions**
  - Can be executed in user mode
- **Sensitive instructions**
  - Can be executed in either kernel or user but they behave differently
  - Require special precautions at execution time.

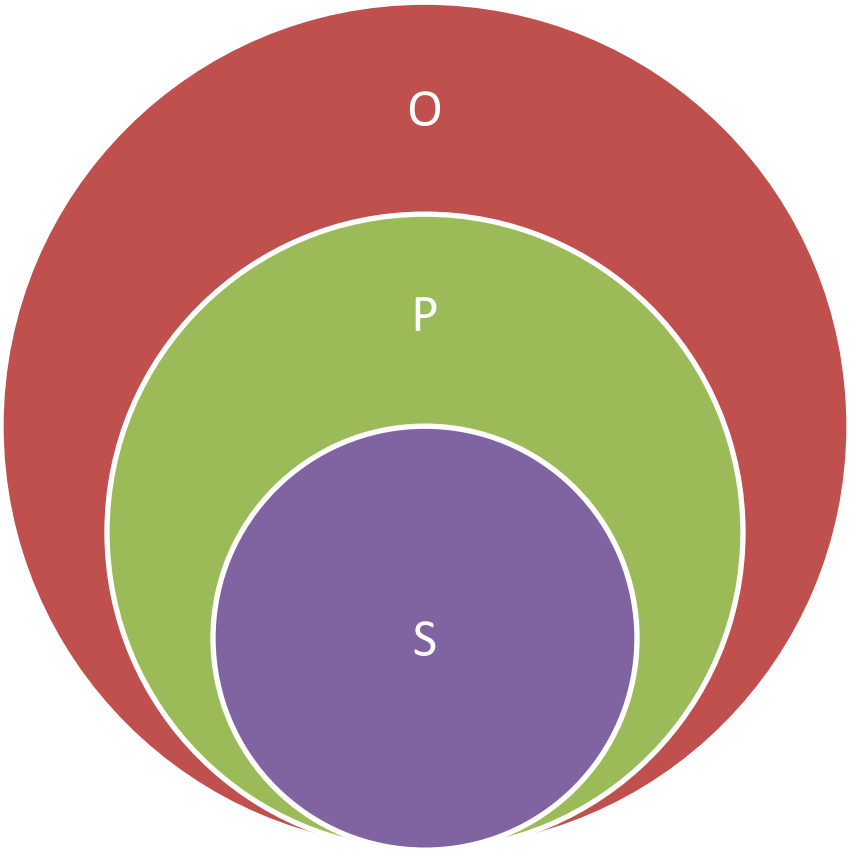
Issues

Let O be the set of instructions, then O = P ∪ S ∪ I, where

- P is the set of **Privileged Instructions**
  - Execution in system mode is possible, execution in user mode traps
- S is a set of **Sensitive instructions**, and S = C ∪ B where
  - C – Control sensitive instructions: Change configuration of system resources
  - B – Behavior sensitive instructions: Behavior depends on configuration of system resources
- I is the set of **Insensitive instructions**

Issues

- Popek and Goldberg theorized that the construction of an efficient VM is possible if $S \subseteq P$
  - All sensitive instructions must also be privileged
- Sadly, this is not the case with the x86 architecture
  - 17 instructions are sensitive but not privileged



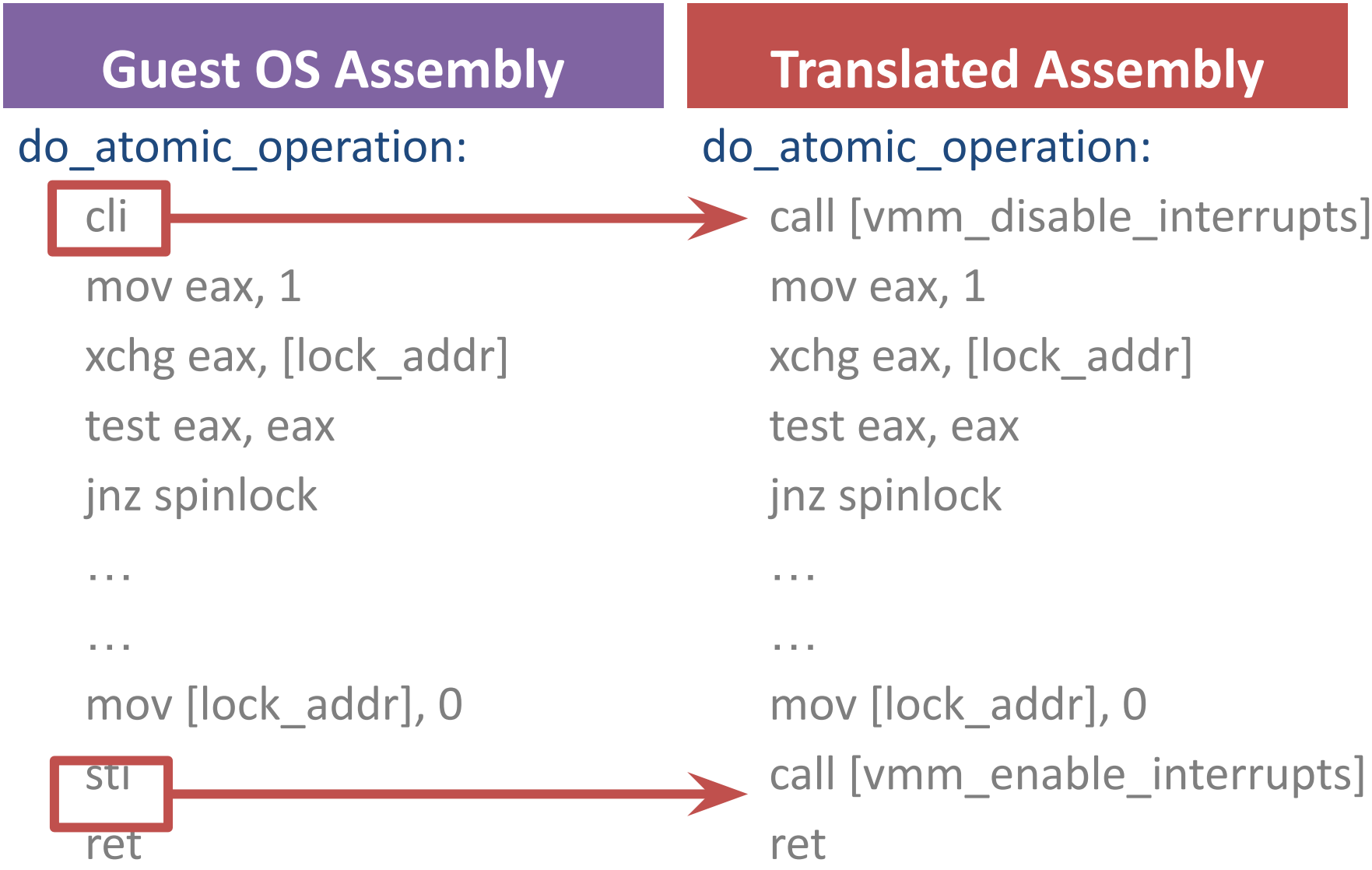
Solutions

- Make sure that no sensitive but unprivileged instructions are executed
  - Instructions are replaced by VMM at runtime
  - Instructions are replaced before runtime
  - Adapting the guest OS
- Hardware becomes virtualization-aware
  - Add new instructions for virtualization
  - Add new mode for virtualization
    - Orthogonal to traditional modes
    - New mode “above” system mode (even more privileged)
  - Example: AMD-V, Intel VT-x, ARM Virtualization Extensions as an extension of ARMv7

Binary Translation

- x86 assembly cannot be virtualized because some privileged instructions don't generate exceptions
- Workaround: translate the unsafe assembly from the guest to safe assembly
  - Known as binary translation
  - Performed by the VMM
  - Privileged instructions are changed to function calls to code in VMM

Binary Translation Example



Pros and Cons

- Advantages of binary translation
  - It makes it safe to virtualize x86 assembly code
  - Translation occurs dynamically, on demand
    - No need to translate the entire guest OS
  - App code running in the guest does not need to be translated
- Disadvantages
  - Translation is slow
  - Wastes memory (duplicate copies of code in memory)
  - Translation may cause code to be expanded or shortened
    - Thus, jmp and call addresses may also need to be patched

Caching Translated Code

- Typically, VMMs maintain a cache of translated code blocks
  - LRU replacement
- Thus, frequently used code will only be translated once
  - The first execution of this code will be slow
  - Other invocations occur at native speed

The Paravirtualized Solution

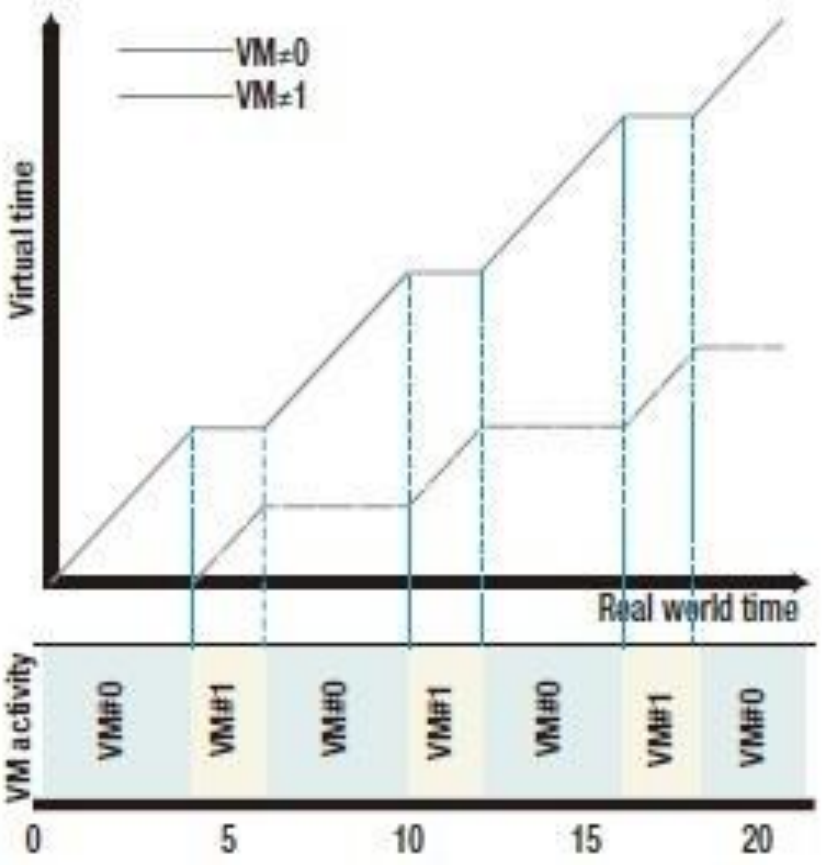
- No privileged instructions are run by the guest OS
  - OS is modified – virtualization aware
  - Executes hypercalls to the hypervisor instead of system calls
  - Hypervisor executes the required privileged instruction safely

Timing in OS

- Maintaining time is crucial for an OS
  - Job deadlines
  - Task switching
- Real time operating systems especially have strict deadlines for tasks
- In a non-virtualized scenario, this is not an issue, as the OS only needs to maintain a single time value – the wallclock time (or real time)

Real vs Virtual Time

- In a virtualized scenario, there are two concepts of time
  - Real time (or wall-clock time)
  - Virtual Time for different OS
- *A virtual machine's virtual time advances only while the VM (here VM#0 or VM#1) is active*



VM Scheduling

- A scheduling algorithm should be efficient, fair, and starvation-free.
  - The objectives of a scheduler for a batch system are to maximize the throughput (the number of jobs completed in one unit of time, e.g., in one hour) and to minimize the turnaround time (the time between job submission and its completion).
  - The objectives of a real-time system scheduler are to meet the deadlines and to be predictable.

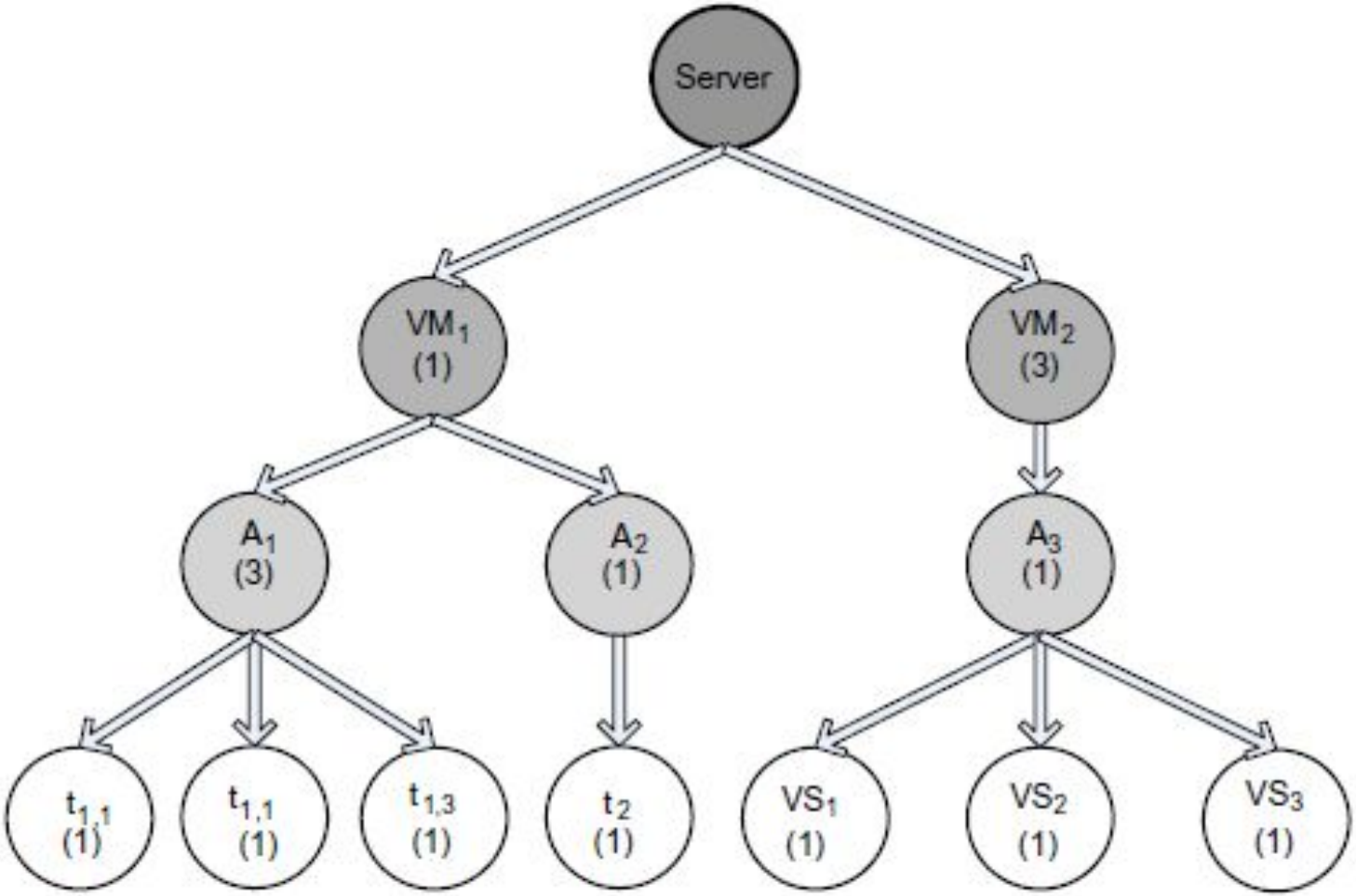
Fairness in Scheduling

- **Max-Min Fairness Criterion**: Consider a resource with bandwidth *B* shared among *n* users who have equal rights. Each user requests an amount *bi* and receives *Bi* . Then, according to the max-min criterion, the following conditions must be satisfied by a fair allocation:
  - The amount received by any user is not larger than the amount requested, $Bi \leq bi$.
  - If the minimum allocation of any user is *Bmin* no allocation satisfying condition *1* has a higher *Bmin* than the current allocation.
  - When we remove the user receiving the minimum allocation *Bmin* and then reduce the total amount of the resource available from *B* to $(B - Bmin)$, the condition 2 remains recursively true.

Start Time Fair Queuing Algorithm

- Hierarchical CPU scheduler
- Organize the consumers of the CPU bandwidth in a tree structure
  - the root node is the processor
  - leaves of this tree are the threads of each application.
  - A scheduler acts at each level of the hierarchy

Start Time Fair Queuing Algorithm



Start Time Fair Queuing Algorithm

- The fraction of the processor bandwidth, *B*, allocated to the intermediate node *i* is $\frac{B_i}{B} = \frac{w_i}{\sum_{j=1}^{n} w_j}$
- When a virtual machine is not active, its bandwidth is reallocated to the other VMs active at the time.
  - When one of the applications of a virtual machine is not active, its allocation is transferred to the other applications running on the same VM.
  - If one of the threads of an application is not runnable, its allocation is transferred to the other threads of the applications.

STFQ Scheduling Rules

- The threads are serviced in the order of their virtual start-up time; ties are broken arbitrarily.
- The virtual startup time of the *i* -th activation of thread *x* is

$$S^{i}{}_{x}(\mathrm{t}) = \max\left[\, \mathrm{v}(\tau^{j}) \,,\, F^{i-1}{}_{x}(\mathrm{t})\right]$$

- The virtual finish time of the *i* -th activation of thread *x* is

$$F^{i}{}_{x}(\mathrm{t}) = S^{i}{}_{x}(\mathrm{t}) + \frac{q}{w_x}$$

- The virtual time of all threads is initially zero. The virtual time *v(t)* at real time *t* is computed as
  - the virtual start time of the thread in service at time *t,* if CPU is busy and
  - the maximum finish virtual time of any thread, if CPU is idle.

Borrowed Virtual Time (BVT) Scheduling

- Objectives of BVT
  - Support low-latency dispatching of real-time applications
  - weighted sharing of the CPU among several classes of applications
  - supports scheduling of a mix of applications, some with hard, some with soft real-time constraints, and applications demanding only a best effort.

Basics of BVT Scheduling

- Thread *i* has
  - *effective virtual time*, E_i
  - *actual virtual time*, A_i
  - *virtual time warp*, W_i
- The scheduler thread maintains its own *scheduler virtual time (SVT)*, defined as the minimum actual virtual time A_j of any thread.
- The threads are dispatched in the order of their effective virtual time, E_i
  - policy called the earliest virtual time (EVT).

Basics of BVT Scheduling

- The virtual time warp allows a thread to acquire an earlier effective virtual time
  - borrow virtual time from its future CPU allocation.
- The EVT of any thread is calculated as

  $EVT = A_i - (warp?\ W_i : 0)$
- The thread with the lowest value of EVT runs
  - The time warp allows real time processes to execute faster

Basics of BVT Scheduling

- Each thread also has a warp time limit L_i and an unwarp time requirement U_i
  - Thread i is allowed to run warped for at most L_i
  - If thread i attempts to warp after having previously warped within U_i , the scheduler runs it unwarped until at least time U_i has passed.
- This prevents the starvation of other processes
- BVT Scheduler is one of the schedulers that was used in Xen hypervisor

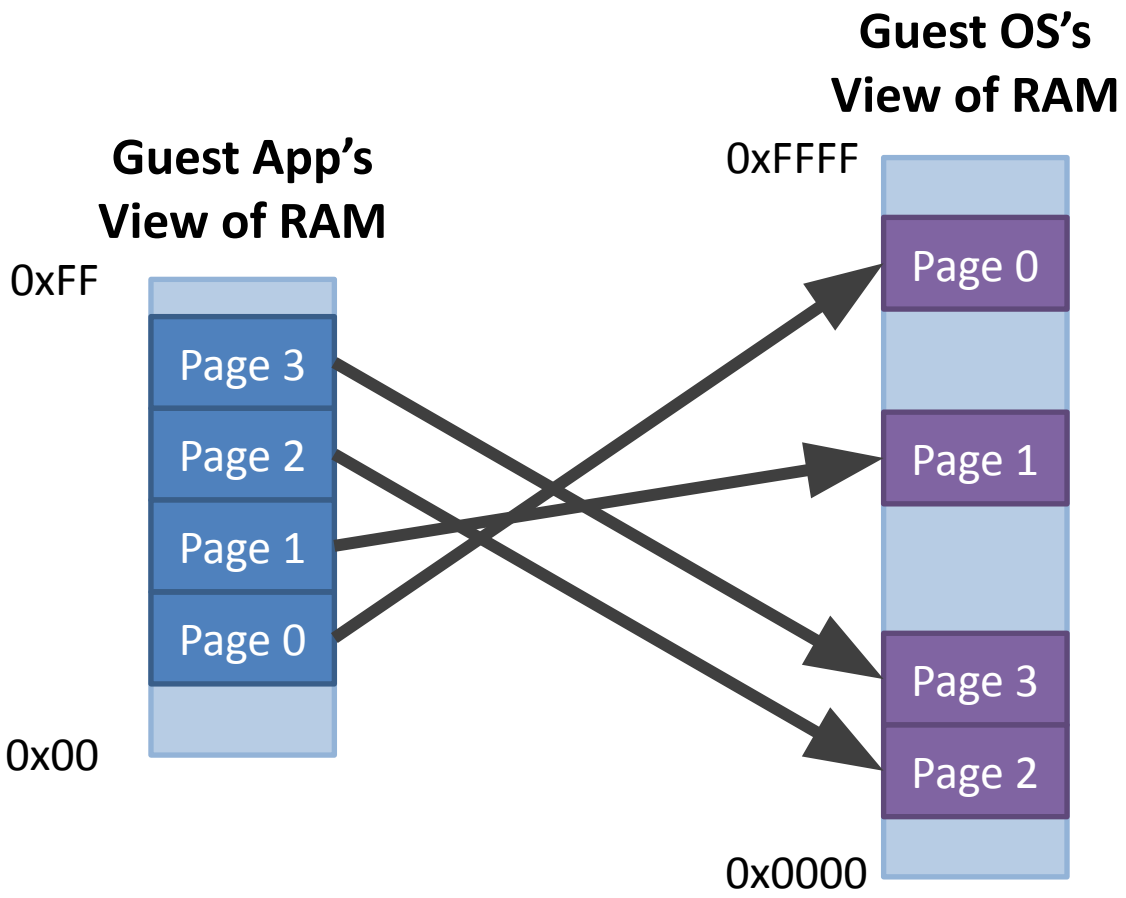
x86 Memory Virtualization

Dr. Amit Praseed

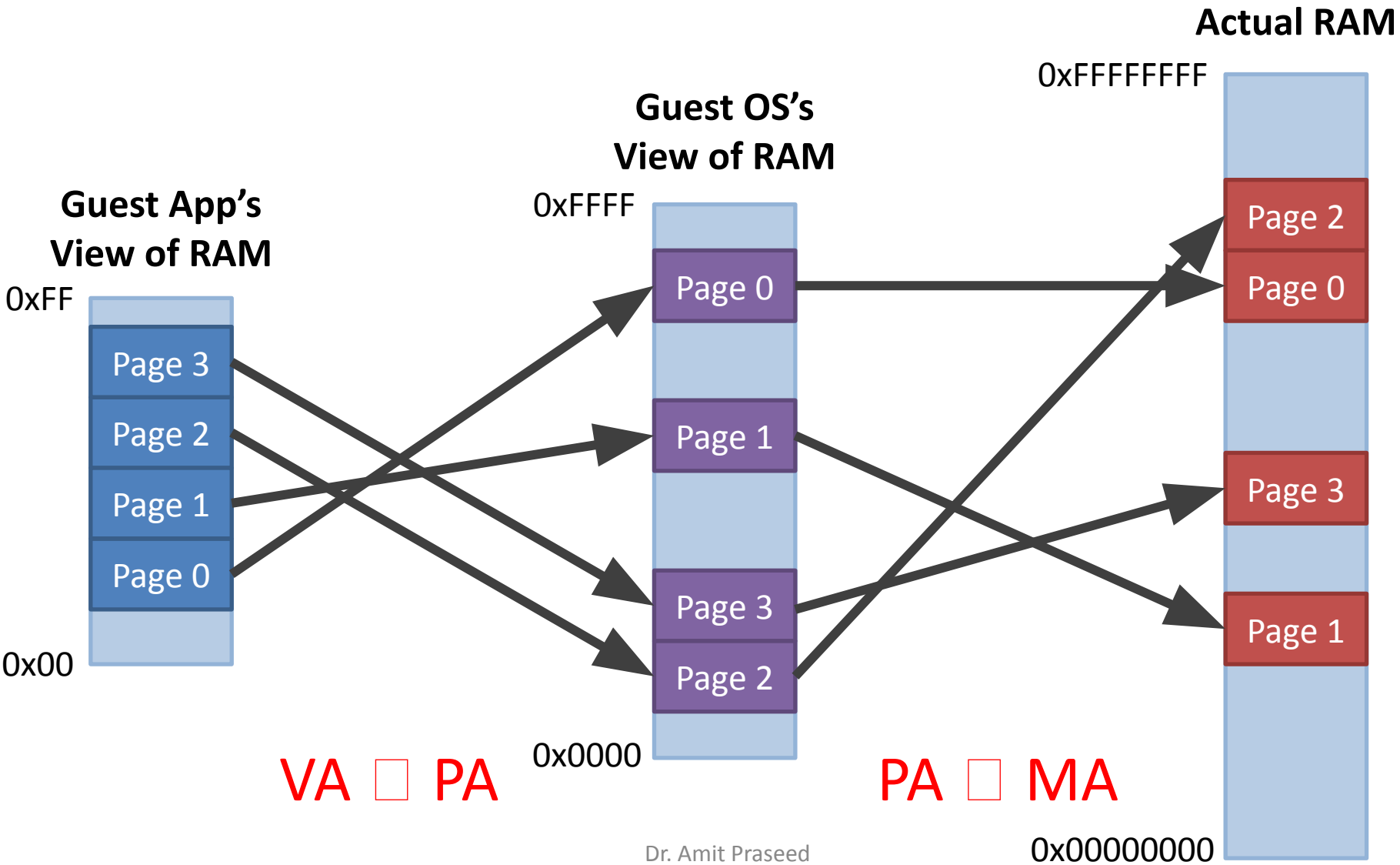
Memory Virtualization

- Typical x86 architecture has a virtual to physical address mapping (VA PA)
- Virtualized x86 architecture requires a two level address translation
  - VA PA
  - Physical address (PA) Machine Address (MA)
- Guest OS has no idea about this translation
  - Guest continues to maintain page tables containing VA PA mappings

Paging without Virtualization



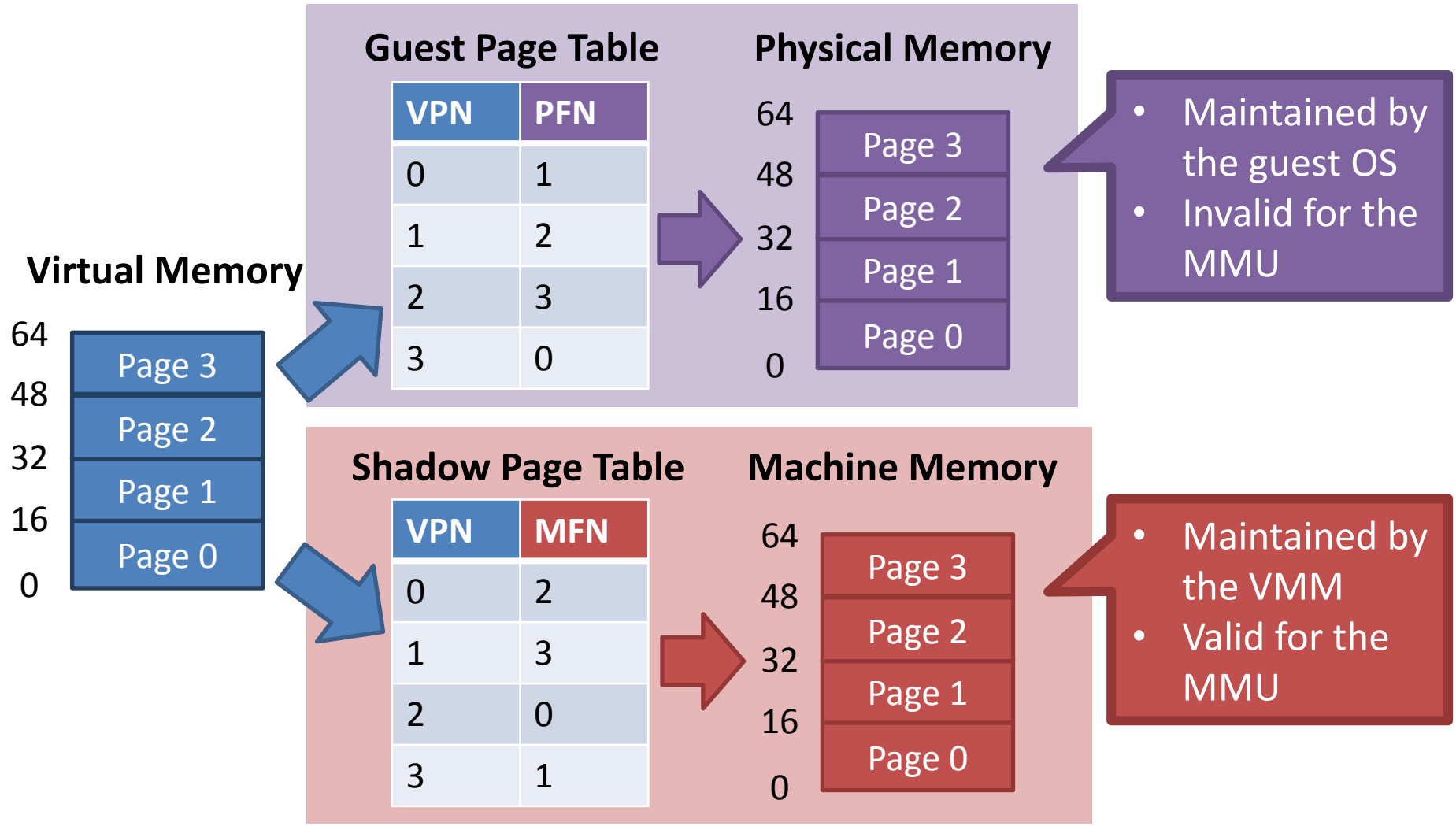
Paging with Virtualization



Does a 2 Level Indirection Work?

- Guest is only aware of VA □ PA mapping
  - Issues only VA to hardware MMU
  - **MMU supports only a single mapping**
- Solution: Shadow Page Table
  - Hypervisor maintains a single shadow page table in the MMU
  - Shadow page table contains direct VA □ MA mapping
  - Trick is to maintain consistency!!!

Shadow Page Tables



Building Shadow Page Tables

- The guest can update its page tables at any time
  - Not a privileged instruction – not trapped!
  - Without knowing when the guest OS updates its page table, the hypervisor cannot maintain the correct entry in the shadow page table
- Solution: Mark the guest page tables as read only
  - Writing generates an exception, which can be trapped by hypervisor

What happens during page faults?

- Two kinds of page faults can occur:
  - True Miss : The mapping does not exist in the guest page table
  - Hidden Miss : The mapping exists in the guest page table, but is absent in the shadow page table
- The hypervisor should disambiguate between the two
- On every miss, the hypervisor walks the guest page table [Tracing]
  - If a mapping exists, the hypervisor silently updates the shadow page table and retries the instruction
  - Otherwise, the hypervisor forwards the page fault to the guest OS for handling

Pros and Cons

- The good: shadow tables allow the MMU to directly translate guest VPNs to hardware pages
  - Thus, guest OS code and guest apps can execute directly on the CPU
- The bad:
  - Double the amount of memory used for page tables
    - i.e. the guest's tables and the shadow tables
  - Overhead due to VMM traps
  - TLB flush whenever a "world switch" occurs
    - Loss of performance

Second Level Address Translation (SLAT)

- Hardware support for memory virtualization
  - Extended Page Tables (EPT) – Intel
  - Nested Page Tables (NPT) - AMD
- Walking the guest and host page tables can be combined into a single multilevel page table
  - Page table depth increases tremendously in some cases!!!
  - Extremely important to use an effective TLB to avoid expensive page table walks
- TLB modified to reduce miss rate
  - Larger TLB
    - More expensive, though!!!
  - Tagged TLB
    - Every entry in TLB now has an address space identifier
    - No need to flush TLB for every “world switch”

I/O Device Virtualization in the x86 Architecture

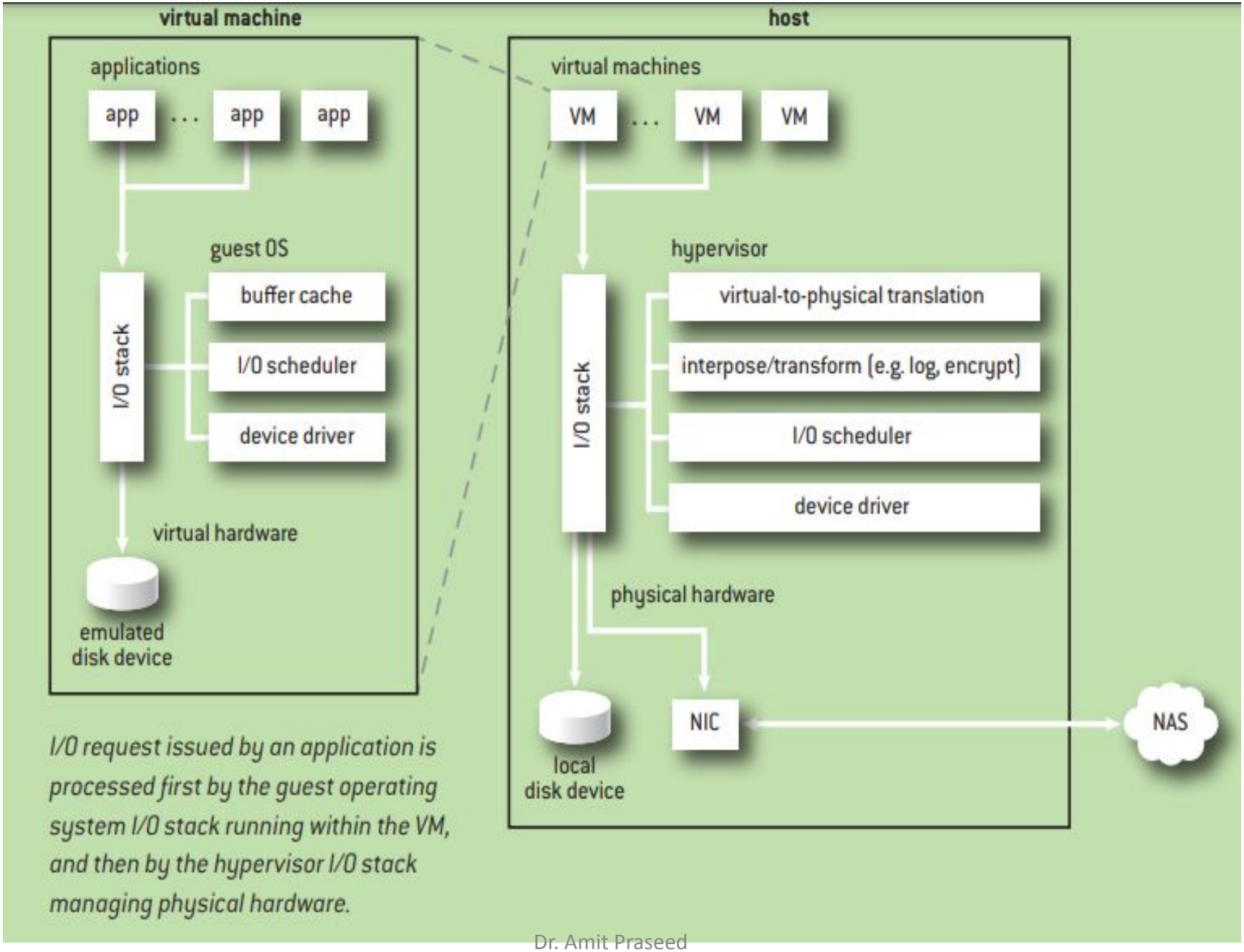
Dr. Amit Praseed

I/O Device Virtualization

- Allows multiple logical devices to be implemented by a smaller number of physical devices
- Provides for flexible mappings between logical and physical devices, facilitating seamless portability
- Device aggregation
  - Multiple physical devices can be combined into a single more capable logical device
- New features can be added to existing systems by interposing and transforming virtual I/O requests, transparently enhancing unmodified software with new capabilities
- Enhanced system security

Challenges

- Defining appropriate semantics for virtual devices and interfaces, especially when faced with complex physical I/O devices or system-level optimizations
- Device contention between multiple VMs
- Achieving good I/O performance despite the potential overhead



I/O request issued by an application is processed first by the guest operating system I/O stack running within the VM, and then by the hypervisor I/O stack managing physical hardware.

Split Driver Architecture

- Most hypervisors employ a split driver architecture for dealing with I/O devices
  - different virtual device interface emulation front-ends
  - multiple different back-end implementations of the device
- The virtualization layer and the virtual-device emulation code must support the architectural and device semantics

Need for Optimization

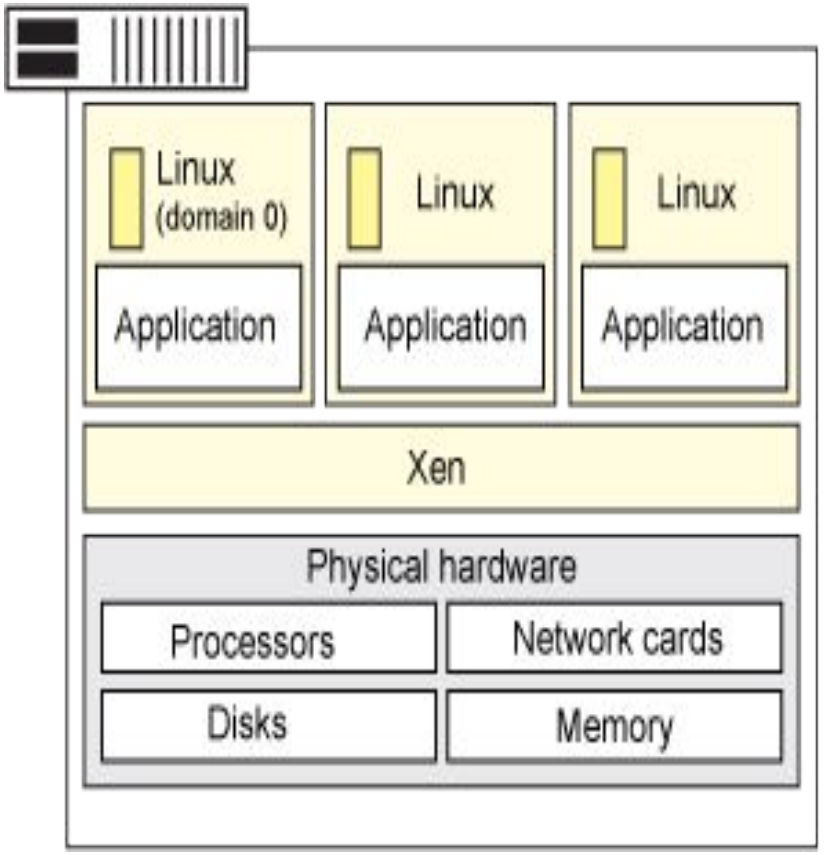
- Reducing I/O-virtualization overhead requires decreasing virtual-device emulation costs
  - reduce the number of trap-and-emulate operations
  - using virtual hardware optimized for the virtualization layer - paravirtualization
  - Pass through mode – when a single VM uses a device

Xen Hypervisor – Case Study

Dr. Amit Praseed

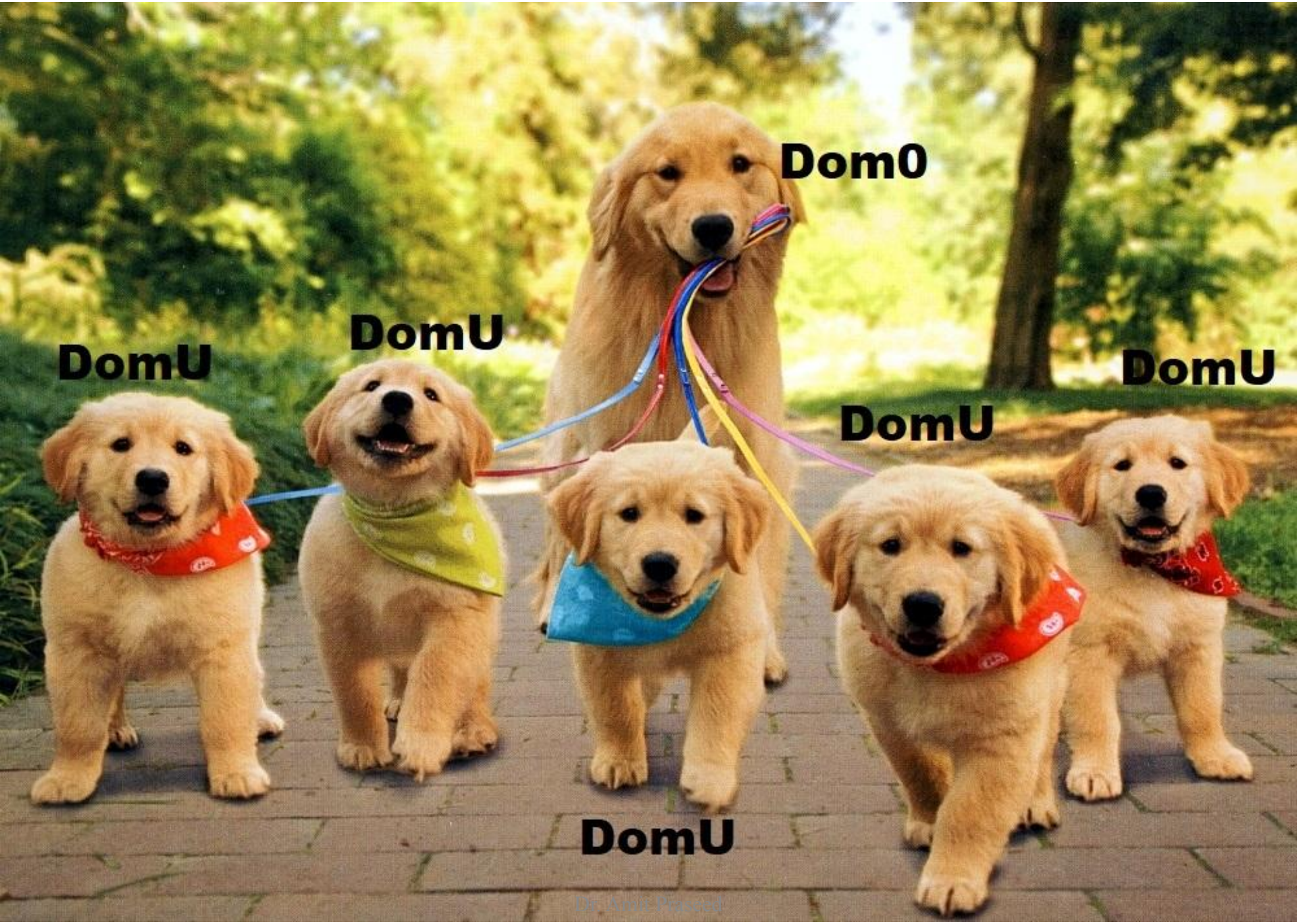
Overview of Xen

- Xen is a type 1 hypervisor
- Creates logical pools of system resources - many virtual machines can share the same physical resources.
- Supports Full and Para virtualization.
- Xen can run multiple guest OS, each in its on VM.



Domain 0

- Xen can run several guest operating systems each running in its own virtual machine or domain.
- When Xen is first installed, it automatically creates the first domain, Domain 0 (or dom0).
  - management domain
  - tasks like building additional domains, managing the virtual devices for each virtual machine, suspending virtual machines, resuming virtual machines, and migrating virtual machines.
- Domain 0 runs a guest operating system and is responsible for the hardware devices.


Dom0
DomU
DomU
DomU
DomU
DomU

A Few Terminologies

- Domain 0 is specially reserved for hypervisor operations
- A general VM is represented as Domain U
  - Paravirtualized VM is represented as Domain U: PV Guests
  - Fully virtualized VM is represented by Domain U: HVM Guests

CPU Virtualization

- Exceptions
  - each domain maintains its own unique and dedicated table of trap handlers - *set_trap_table*
  - System calls are carried out at near-native speeds without having to be executed in Ring-0.
  - Page Fault exceptions are carried out by Xen and the register values stored for retrieval by the guest in Ring-1.
- Scheduling
  - Borrowed Virtual Time (BVT) scheduling scheme
  - Hybrid algorithm that is both work-conserving and has mechanisms for low-latency dispatch, or domain wake-up

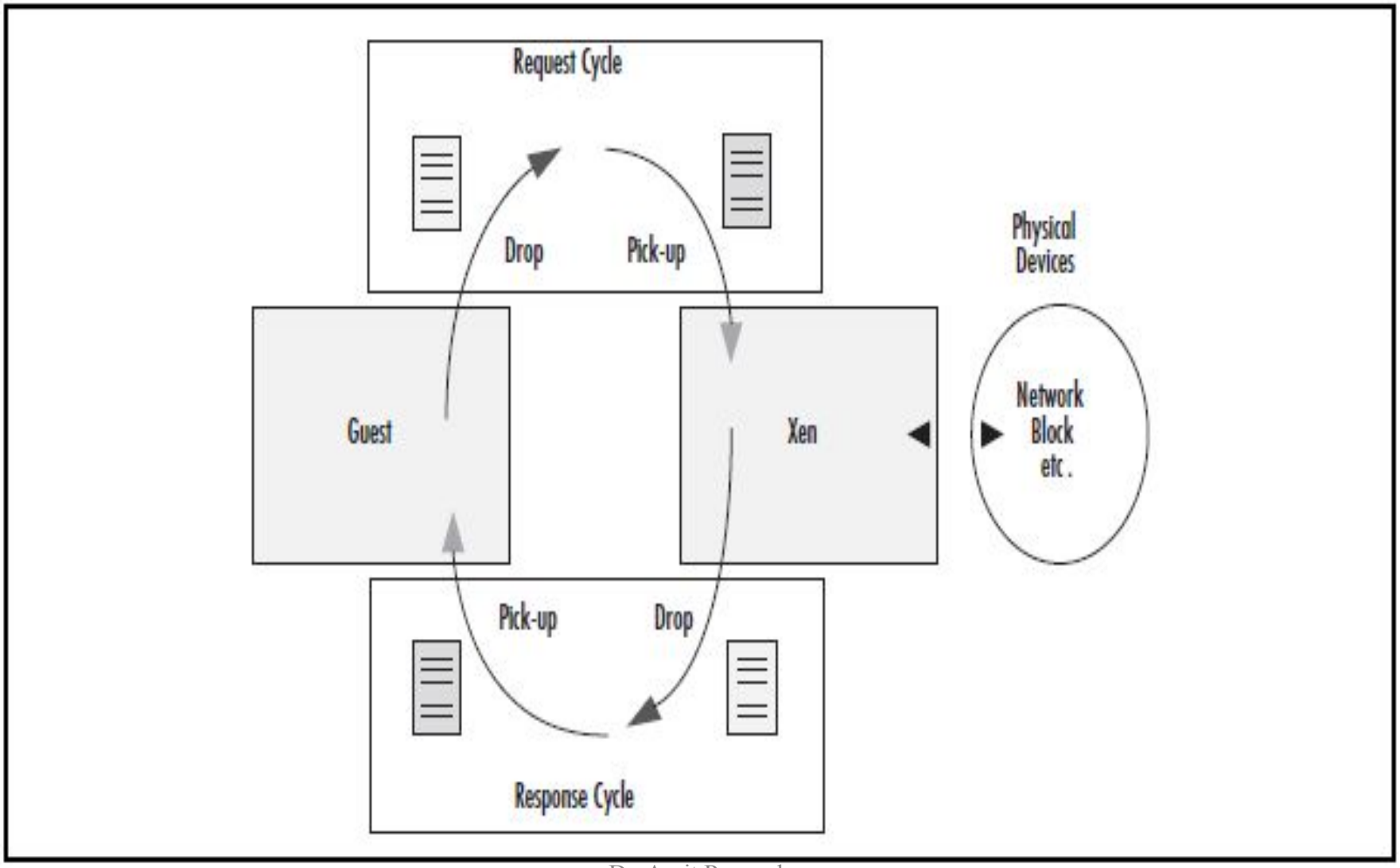
Memory Virtualization

- In a 32 bit architecture, Xen reserves the top 64 MB space for itself
- Each domU has a maximum and current consumed physical memory allocation.
  - Balloon driver concept for each domain, that allows the operating system to adjust its current memory allocation
  - Allows “unused” allocation to be consumed in other areas
- Shadow Page Tables for memory virtualization
  - Can use SLAT if hardware virtualization is enabled

I/O Virtualization

- Xen follows a paravirtualization approach for I/O virtualization
  - installing device drivers that are coded to be Xen-aware and interact with the hypervisor APIs
  - Can be used by both modified/unmodified OS
- Three components cooperate to allow for device virtualization in Xen
  - Device I/O Rings
  - Event Channels
  - Virtual I/O Devices and Split Device Drivers

Device I/O Rings



Event Channels

- Event channels are mediums through which events, the Xen equivalent of a hardware interrupt, are transmitted.
- Event notifications are received via an upcall from Xen.

Split Device Drivers

- When Xen boots and launches dom0, it exports a subset of the devices in the systems to the other domains
  - based on each domain's configuration
  - *Class devices,* not as a specific hardware model
- This architecture comprises two cooperating drivers:
  - the *frontend driver* in an unprivileged domU,
  - the *backend driver*, which runs in dom0

Split Device Drivers

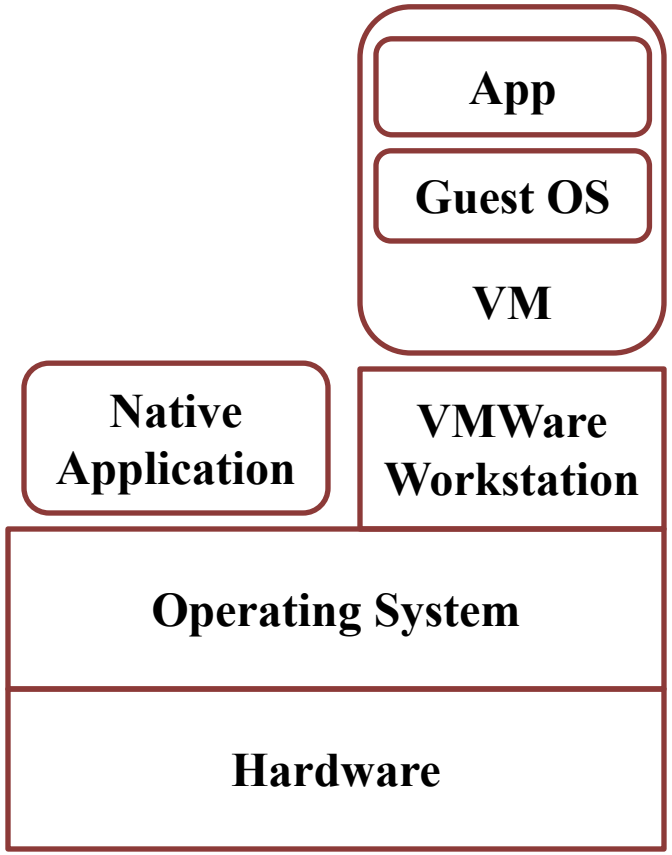
- The frontend driver appears to the domU guest operating system as a real device.
  - The guest can interact with it just as it would any other device for which it had the appropriate drivers installed.
  - It can receive I/O requests from its kernel, but since it does not have direct access to the hardware, it must pass those requests to the backend driver running in dom0.
- The backend driver
  - receives the I/O requests,
  - validates them for safety and isolation
  - proxies them to the real device.
  - When the I/O operation completes, the backend driver notifies the frontend driver

VMWare Workstation Hypervisor – Case Study

Dr. Amit Praseed

Overview of VMWare Workstation

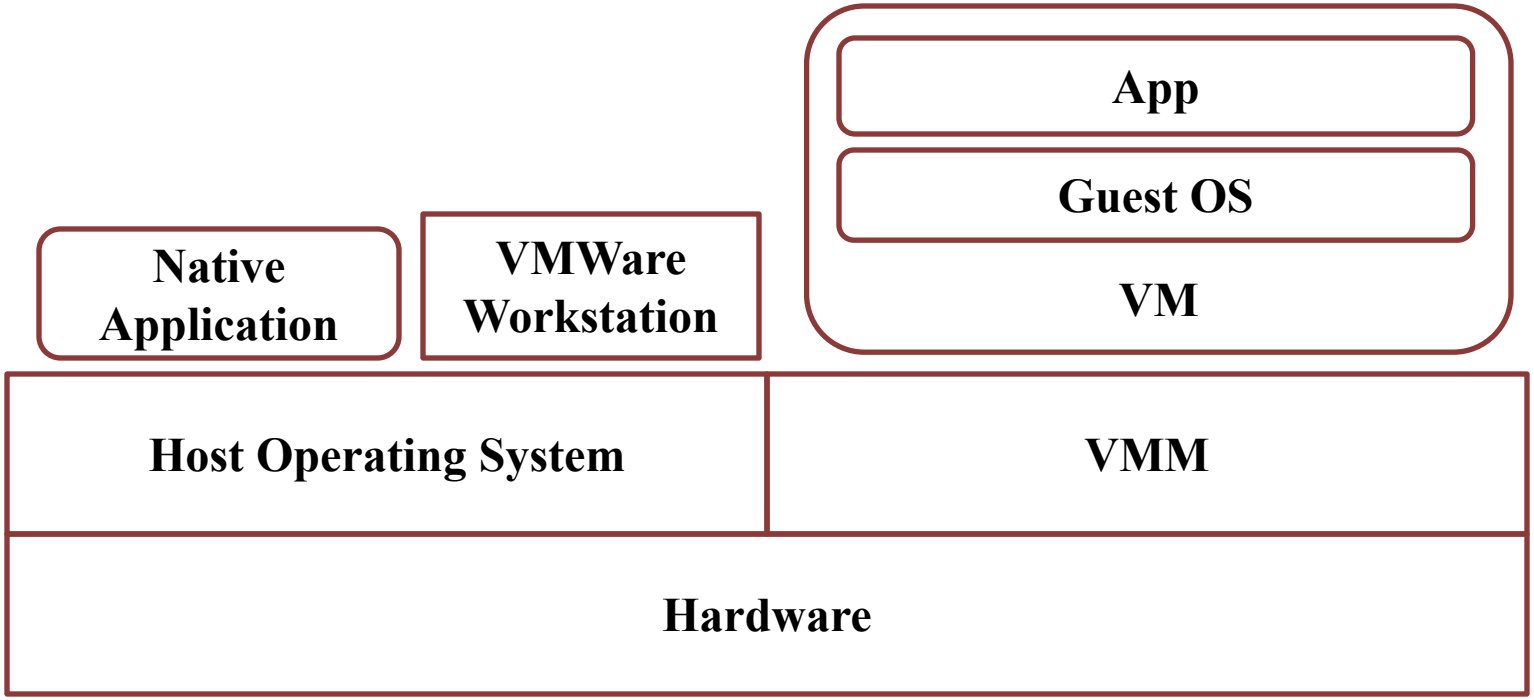
- VMWare Workstation is a type 2 (hosted) hypervisor
  - Runs on a host operating system
- Design Goals
  - Provide the traditional benefits of virtual machines
  - Virtualize x86 PCs
  - Run like an application
  - Run with good performance



Requirements of VMWare

- Virtualization techniques require
  - Access to privileged CPU state
  - Complete control of the Memory Management Unit (MMU)
- The Host OS gets in our way
- **Need to both run as an application and as a privileged virtual machine monitor**

3 – Component Model



- Switch to privileged VMM to run virtual machine
  - VMM takes complete control of CPU and MMU
  - Host OS state is saved / restored on switch

Resource Usage by VMWare

- Host OS assumes it is in control all the time
- VMM takes control for a bounded amount of time
- Each VM is controlled by a distinct VMM instance
- Distinct process instance of host OS – VMX
- Kernel Resident Driver: Responsible for locking memory pages, forwarding interrupts, world switch etc.
- Interrupts are handed over to the host OS for handling

CPU Virtualization

- Traditional virtualization technique
  - Direct Execution
  - Run OS at an unprivileged CPU level
  - CPU traps to VMM on privileged instructions
  - VMM emulates privileged instruction
- However, x86 is not strictly virtualizable (Why?)

Solution : Binary Translation

- Inspects each instruction before its executed
- Replaces “dangerous” instructions with calls to emulation code
- Stores sequences of translated instructions in a translation cache
- Fast, but slower than direct execution

Dual Virtualization Method

- Use traditional direct execution when possible
  - Well behaved user-level programs
- Use binary translation when not
  - Operating system
  - Real mode programs (old 16 bit DOS apps)
  - User-level programs with special privileges

Proportional Share (Lottery) Scheduling

- One very basic concept: tickets
  - Represent the share of a resource that a process should receive.
  - The percent of tickets that a process has represents its share of the system resource in question
- Achieved probabilistically by holding a lottery every so often
- Holding a Lottery:
  - the scheduler must know how many total tickets there are
  - Scheduler picks a winning ticket
  - Whichever process holds the ticket wins
  - Probabilistic correctness in meeting the desired proportion, but no guarantee

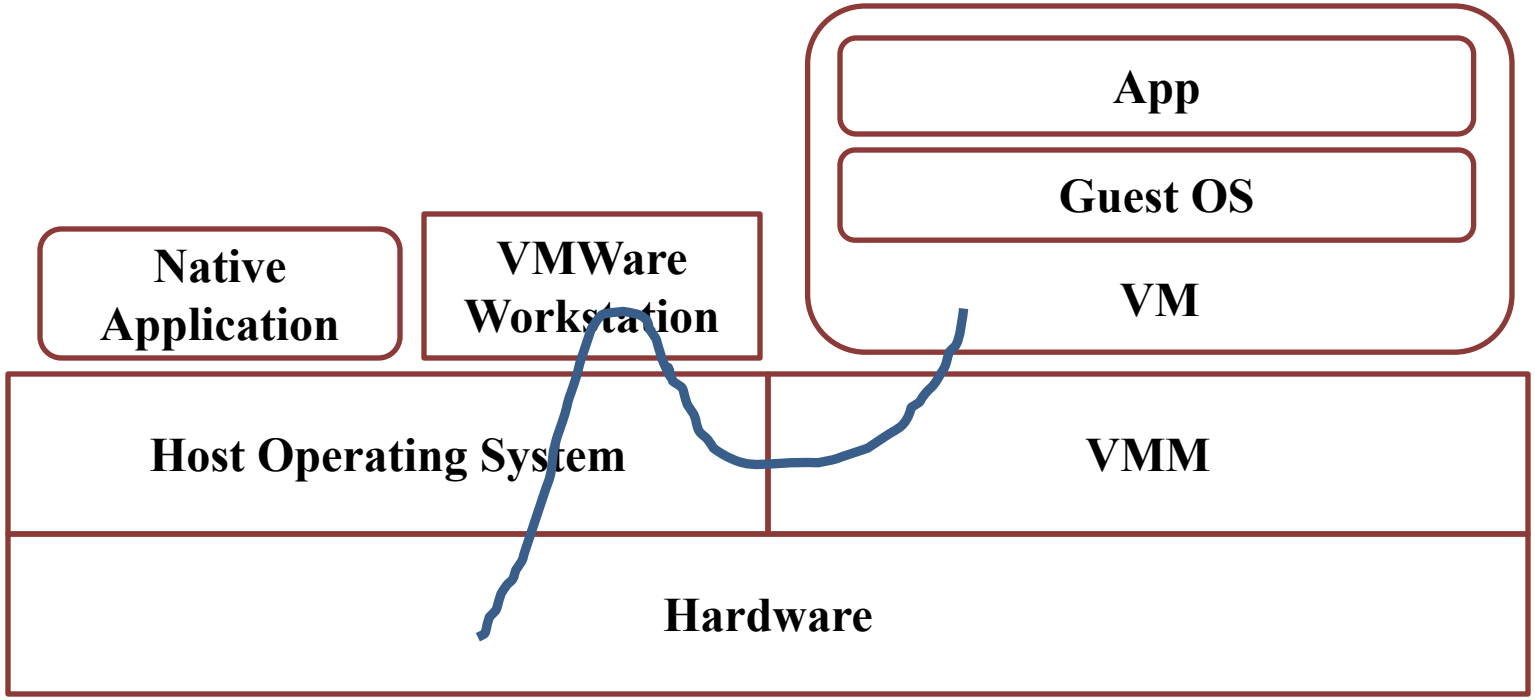
Memory Management

- Protecting the VMM:
  - The VMM address space needs to be protected from other VMs
  - Segment Truncation
    - Only restricts the limit – hence VMM needs to be in the top 4 MB of memory
- Host physical memory is managed by VMX process
  - Locks required pages in memory
  - Shadow Paging to ensure that guest OS only request valid pages

Device Management

- Hosted hypervisors can utilize the device drivers in the underlying OS
  - I/O requests are remote procedure calls between VMM and VMX
  - VMX issues system calls to the host OS
  - Interrupts generated within the guest OS are handed over to the host OS
  - Executed as if VMX thread has issued the interrupt

Device Usage by VMWare

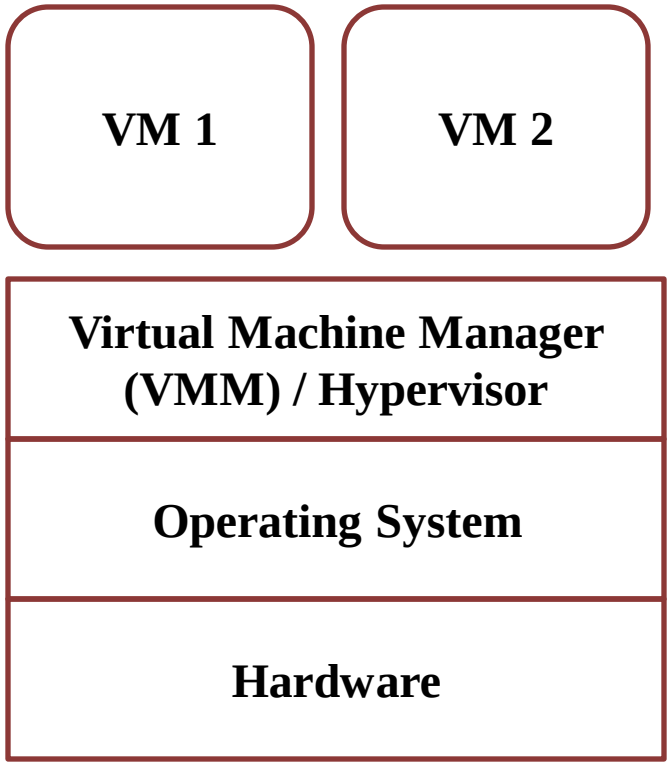


- Device I/O is routed through the application
- Use standard OS system call interfaces

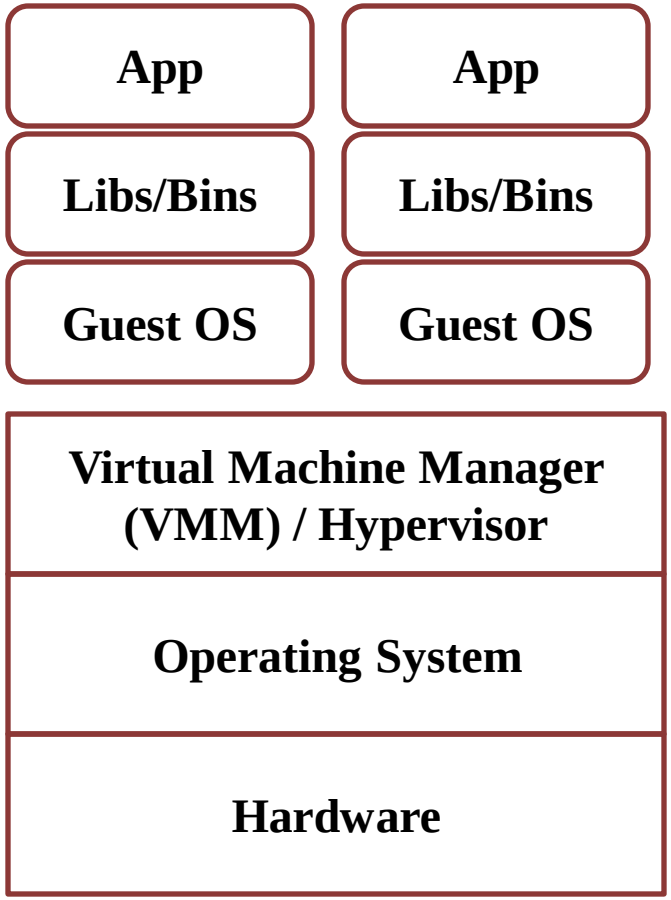
Containers

Dr. Amit Praseed

Review of Virtual Machines



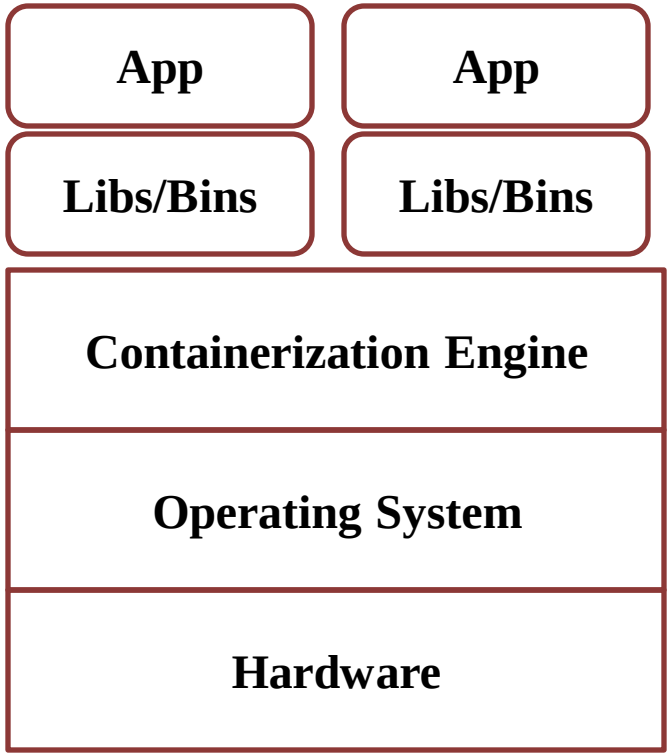
Review of Virtual Machines



What is a Container?

- Containerization involves
  - bundling an application together with all of its related configuration files, libraries and dependencies
  - run in an efficient and bug-free way across different computing environments
- Solves a major problem in software development
  - “It works on my system”
  - Containers bundle all related files and libraries with the application, so it can run anywhere

How are Containers Organized?



Containerization in One Image

Containers vs VMs

Containerization

- Isolates and encapsulates a single application
- Lightweight
- Migration is very simple

- OS Level Virtualization
- *Technically* can only run on same / similar underlying OS as the container

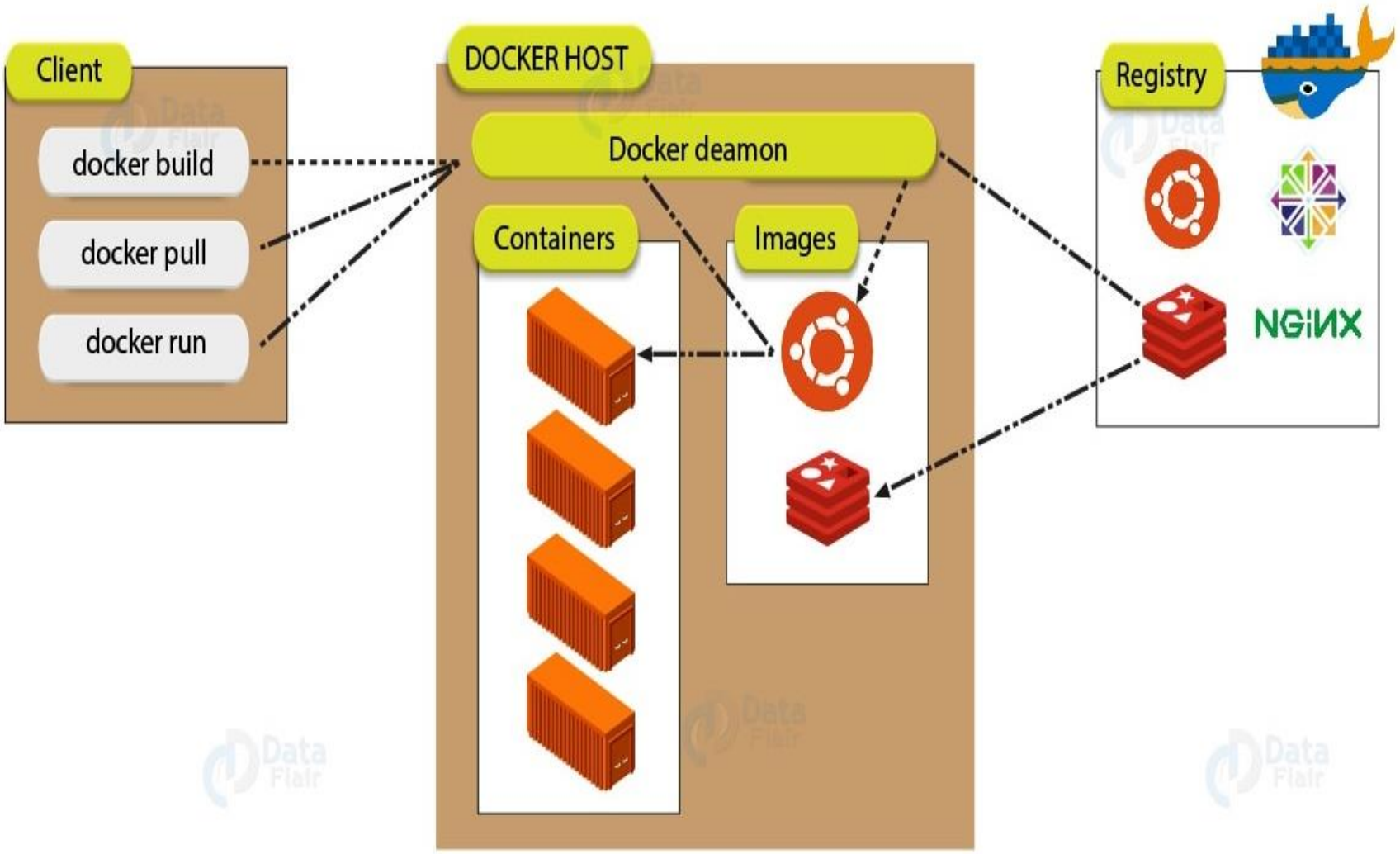
Virtual Machines

- Isolates and encapsulates an entire OS
- Heavyweight
- Migration is comparatively difficult
- Hardware level virtualization
- Can be run on any underlying OS with proper hypervisor

Docker

- Docker is an open-source engine that automates the deployment of applications into containers
  - adds an application deployment engine on top of a virtualized container execution environment.
  - provides a lightweight and fast environment
- Features of Docker
  - An easy and lightweight way to model reality
  - A logical segregation of duties
  - Fast, efficient development life cycle
  - Encourages service orientated architecture

Docker Architecture



Basics of Docker

- Docker Images : Images are the “source code” of containers. It encompasses a particular application environment
- Registry : Docker images can be published to public or private registries, so they can be easily reused
- Docker Containers : A container is a live instance of a Docker image

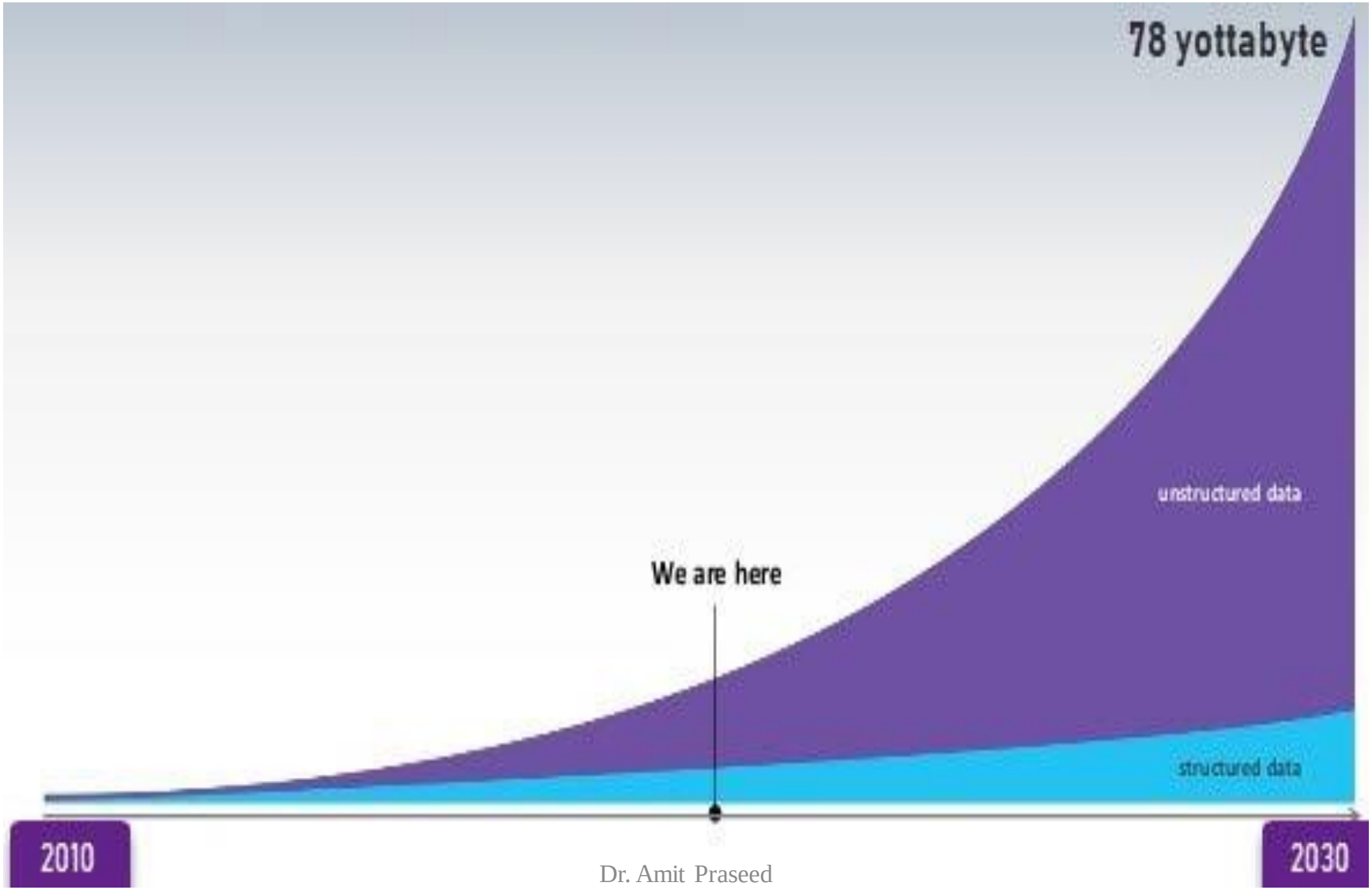
Big Data and its Implications on the Cloud

Dr. Amit Praseed

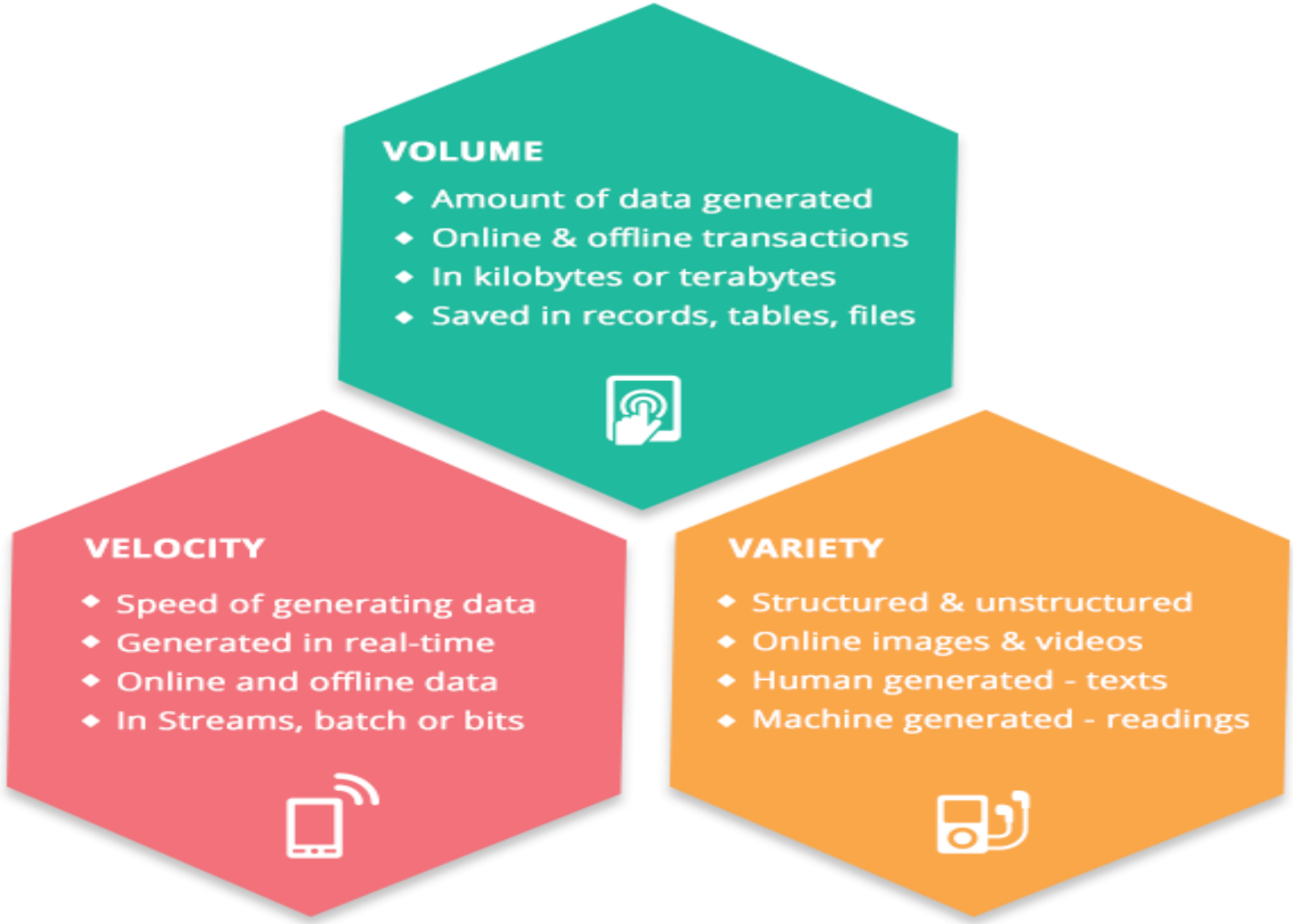
How much data do we generate?

- **Structured Data**
  - Relational Databases
  - Well defined schema
- **Unstructured Data**
  - Videos, audio, images etc.
- **Semi-structured Data**
  - structured in form but not well defined (no schema)
  - XML files

The Rise of Unstructured Data



THE 3Vs OF BIG DATA



The Rise of NoSQL

- NoSQL = Not only SQL
- A fundamental shift or alternative to storing data which does not conform to the relational format
- Features
  - Schema Agnostic
  - Non relational
  - Commodity hardware
  - Highly distributable

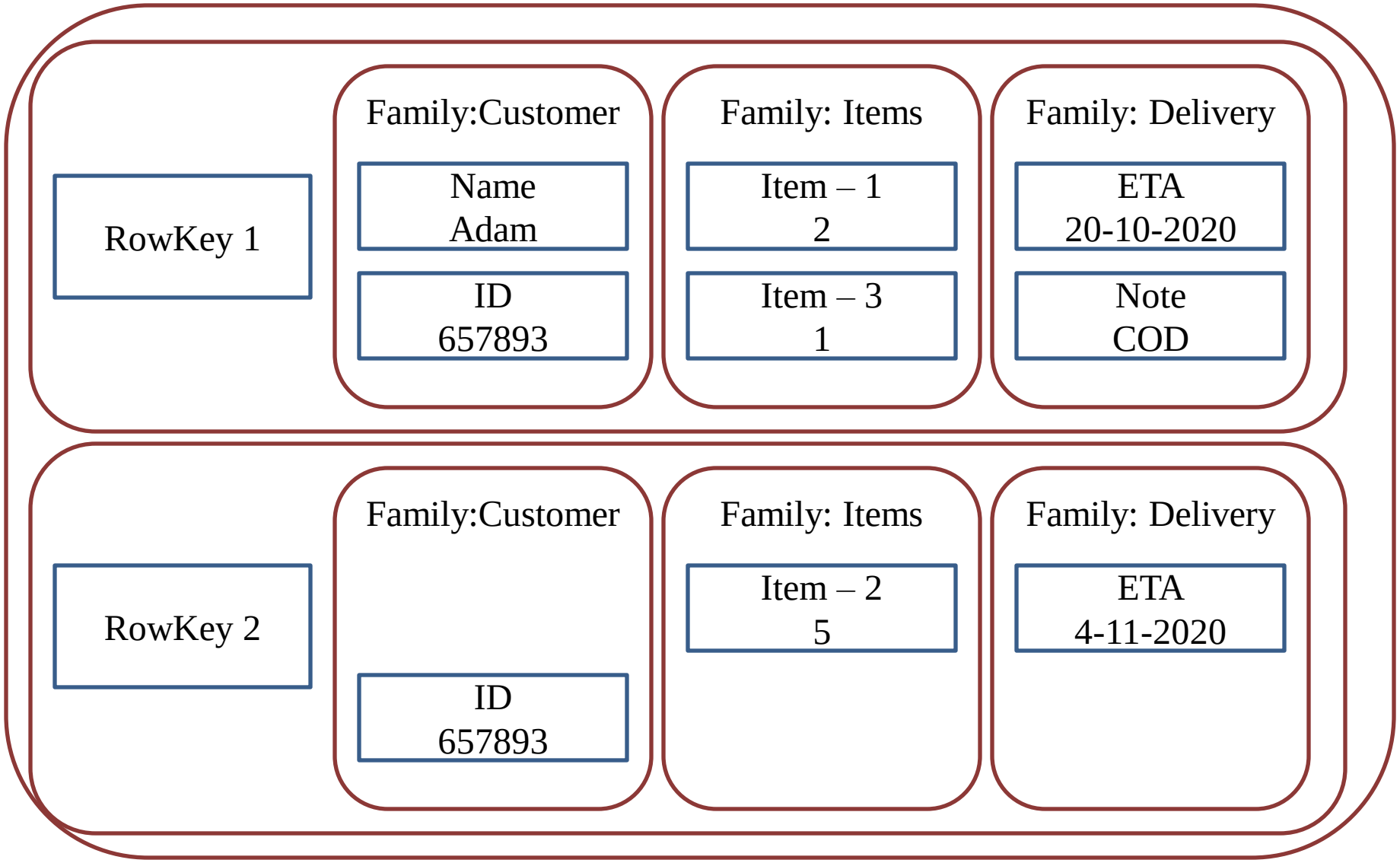
Four Core NoSQL Varieties

- Columnar
- Key-Value
- Triple
- Document

Columnar Databases

- Similar to relational model – concept of rows and columns still exist
- Optimized and designed for faster column access
- Ideal for running aggregate functions or for looking up records that match multiple columns
- A single record may consist of an ID field and multiple column families
- Each one of these column families consists of several fields. One of these column families may have multiple “rows”

Columnar Databases



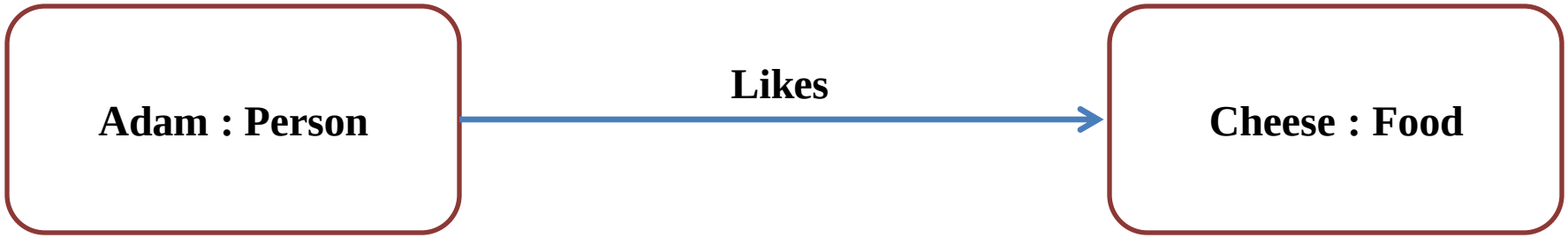
Key – Value Stores

- An ID field — the key in key-value stores — and a set of data
- Some key-value stores support typing (such as integers, strings, and Booleans) and more complex structures for values (such as maps and lists)
- Key-value stores are optimized for speed of ingestion and retrieval

Triple and Graph Stores

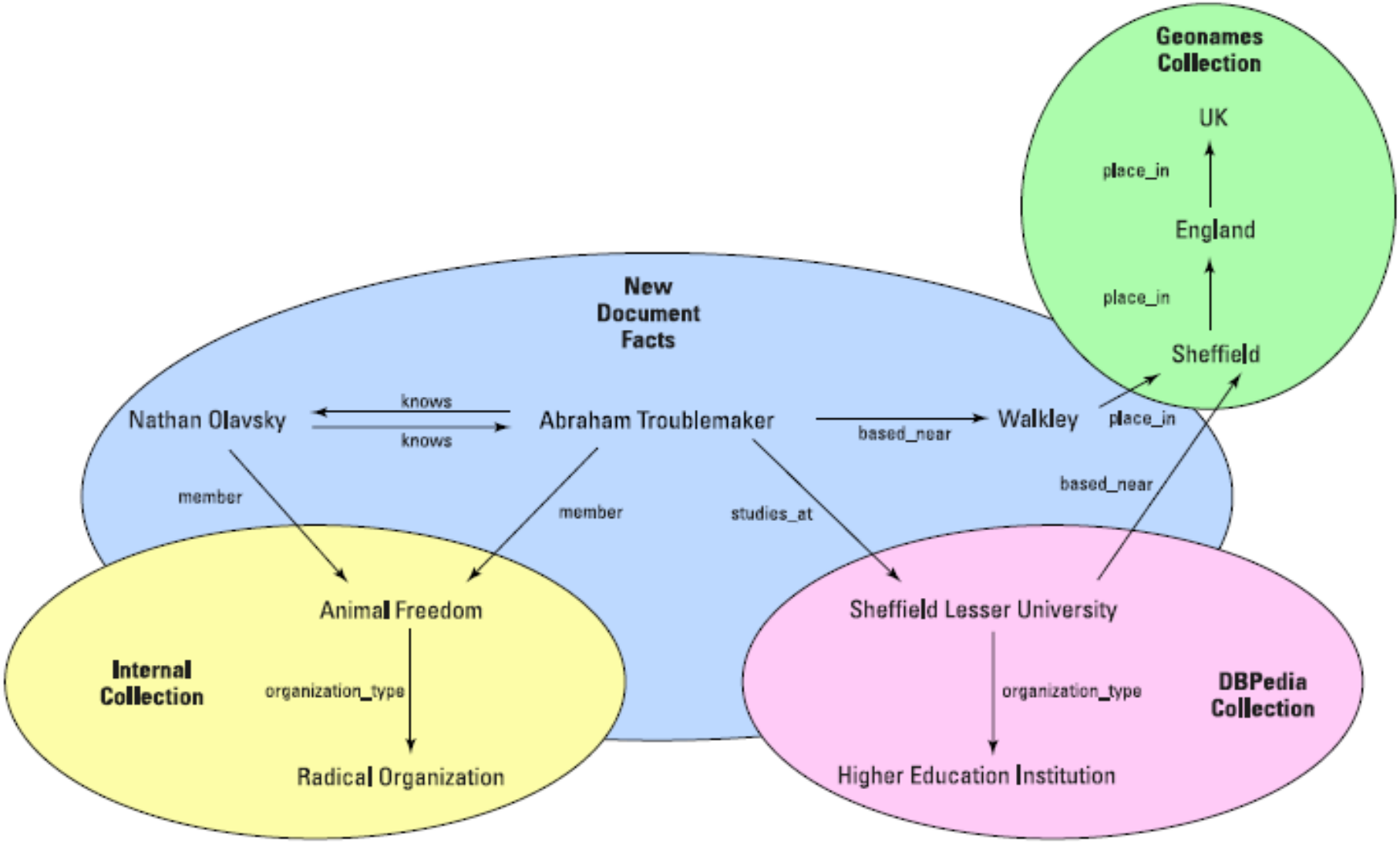
- Every *fact* or assertion is described as a triple of subject, predicate, and object:
  - A *subject* is the thing you're describing. It has a unique ID called an IRI. It may also have a type, which could be a physical object (like a person) or a concept (like a meeting).
  - A *predicate* is the property or relationship belonging to the subject. This again is a unique IRI that is used for all subjects with this property.
  - An *object* is the intrinsic value of a property (such as integer or Boolean, text) or another subject IRI for the target of a relationship.

Triple and Graph Stores



- Three points of information
  - Adam is a person
  - Cheese is a food item
  - Adam likes cheese

Triple and Graph Stores



Document Stores

- Hold documents that combine information in a single logical unit
- Retrieving all information from a single document is easier with a database and is more logical for applications
- A document is any unstructured or tree-structured piece of information.
  - It could be a recipe, financial services trade, PowerPoint file, PDF, plain text, or JSON or XML document.

Examples of NoSQL Products

- **Columnar:** DataStax, Apache Cassandra, HBase, Apache Accumulo, Hypertable
- **Key-value:** Basho Riak, Redis, Voldemort, Aerospike, Oracle NoSQL
- **Triple/graph:** Neo4j, Ontotext's GraphDB (formerly OWLIM), MarkLogic, OrientDB, AllegroGraph, YarcData
- **Document:** MongoDB, MarkLogic, CouchDB, FoundationDB, IBM Cloudant, Couchbase

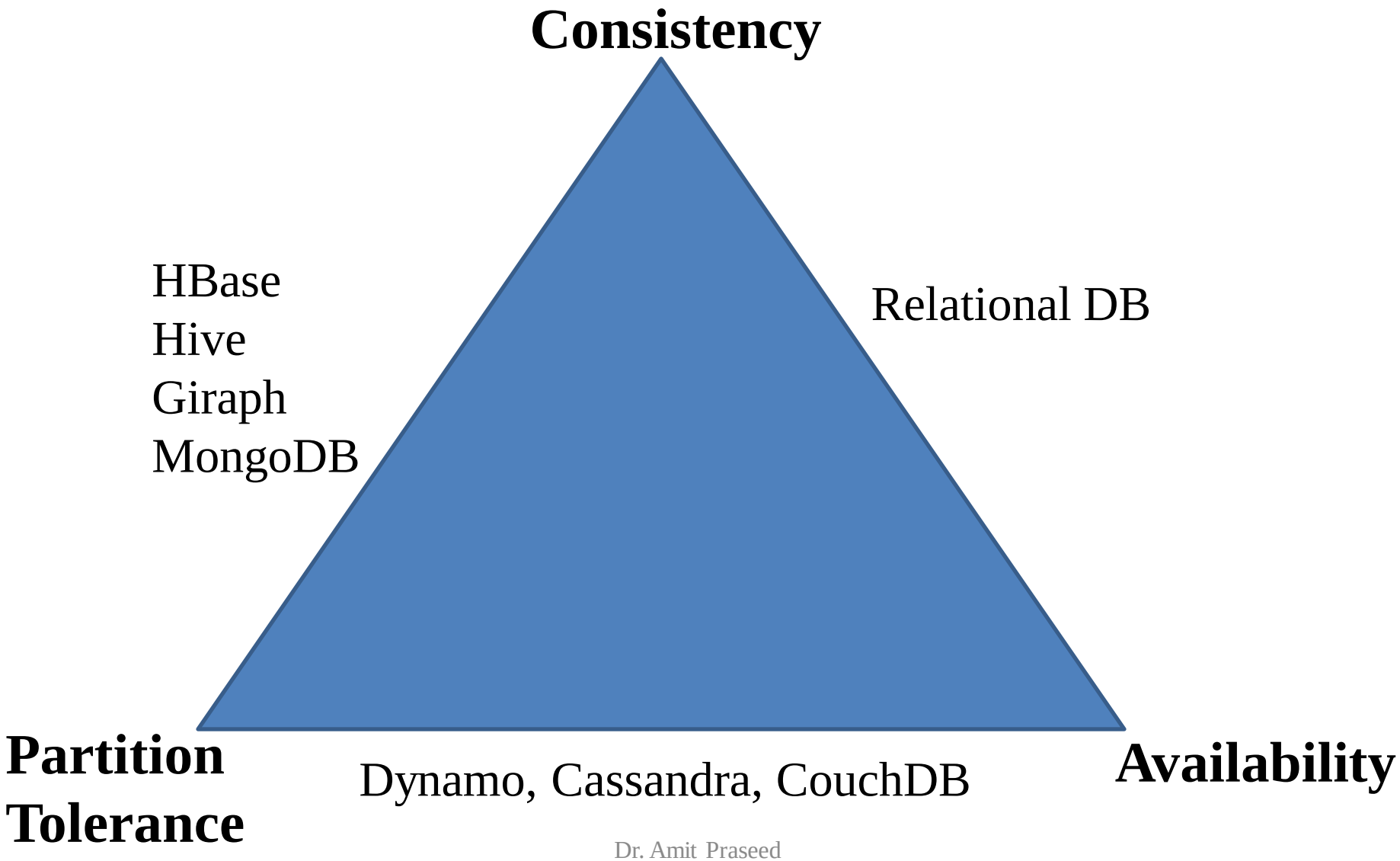
The CAP Theorem

- It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:
  - **Consistency**: Every read receives the most recent write or an error
  - **Availability**: Every request receives a response, without the guarantee that it contains the most recent write
  - **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped by the network between nodes

Consistency in NoSQL

- NoSQL relaxes the ACID consistency model used in relational databases
- NoSQL uses an eventual consistency model called BASE
  - Basically Available
  - Soft state
  - Eventual consistency

ACID, BASE and the CAP Theorem



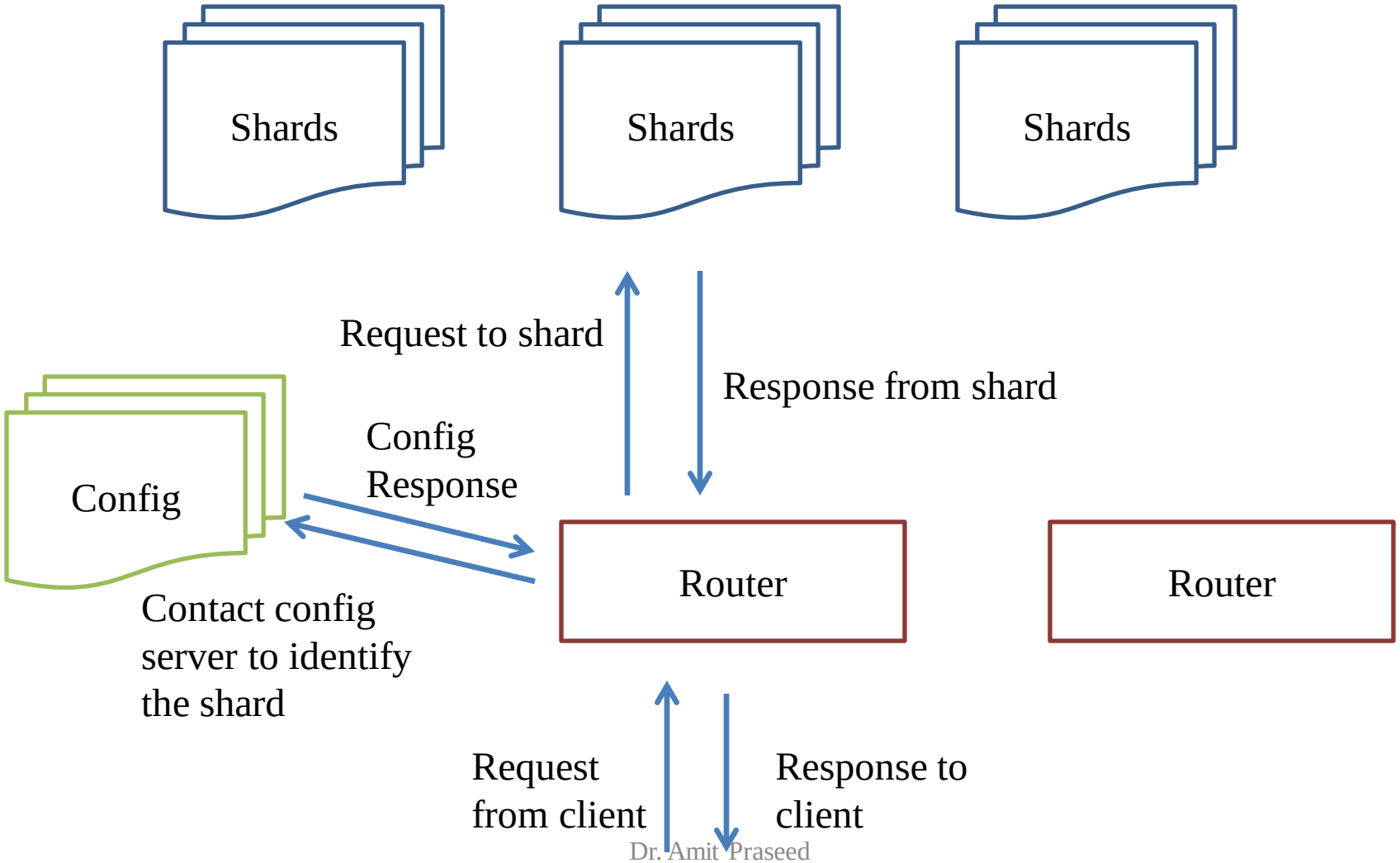
MongoDB

- A document-oriented, NoSQL database
  - Hash-based, schema-less database
  - Atomic writes and fully-consistent reads
  - Master-slave replication with automated failover (replica sets)
  - Built-in horizontal scaling via automated range-based partitioning of data (sharding)
  - No joins nor transactions
  - Consistency + Partition Tolerance

MongoDB Terminology

- A MongoDB instance may have zero or more ‘databases’
- A database may have zero or more ‘collections’
- A collection may have zero or more ‘documents’
- A document may have one or more ‘fields’
- MongoDB ‘Indexes’ function much like their RDBMS counterparts.

MongoDB Architecture

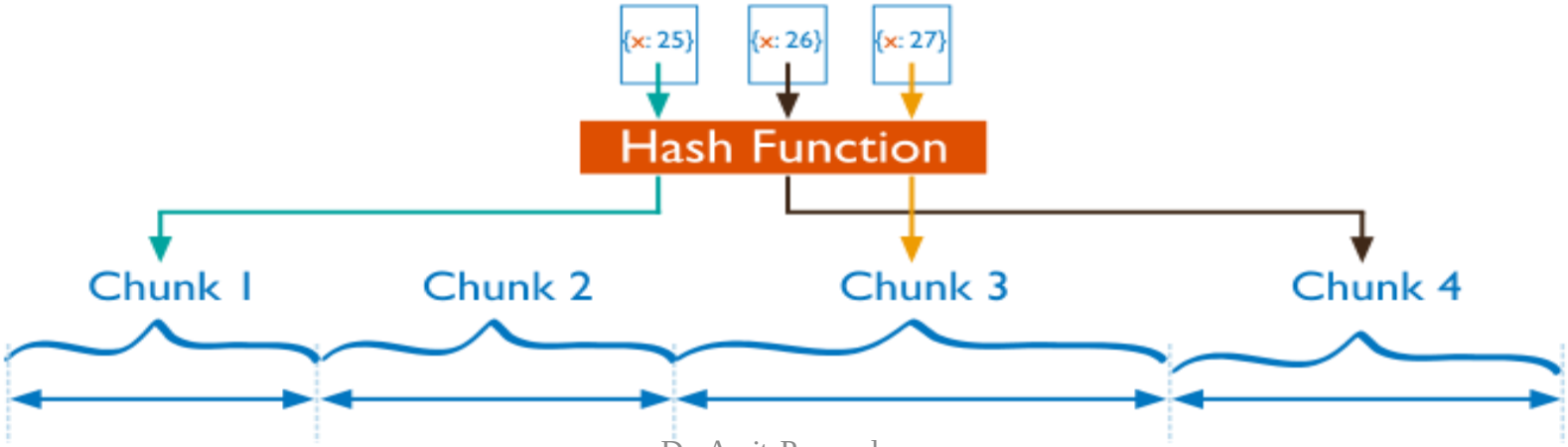


MongoDB Replication

- A replica set is a group of mongod instances that maintain the same data set.
  - A replica set contains several data bearing nodes and optionally one arbiter node.
  - Of the data bearing nodes, one member is deemed the primary node, while the other nodes are deemed secondary nodes.
- The primary node receives all write operations.
  - The primary records all changes to its data sets in its operation log, i.e. oplog
  - The secondaries replicate the primary's oplog and apply the operations to their data sets

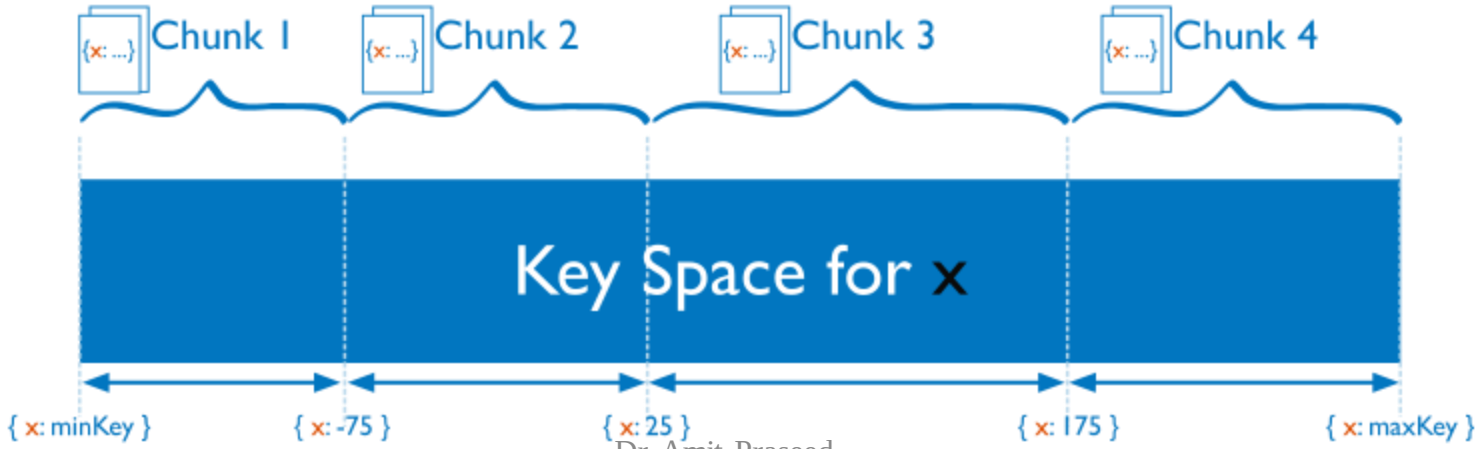
Sharding in MongoDB

- **Hashed Sharding**
  - Involves computing a hash of the shard key field's value.
  - Each chunk is then assigned a range based on the hashed shard key values.
  - Even data distribution, but may result in more cluster wide broadcast operations



Sharding in MongoDB

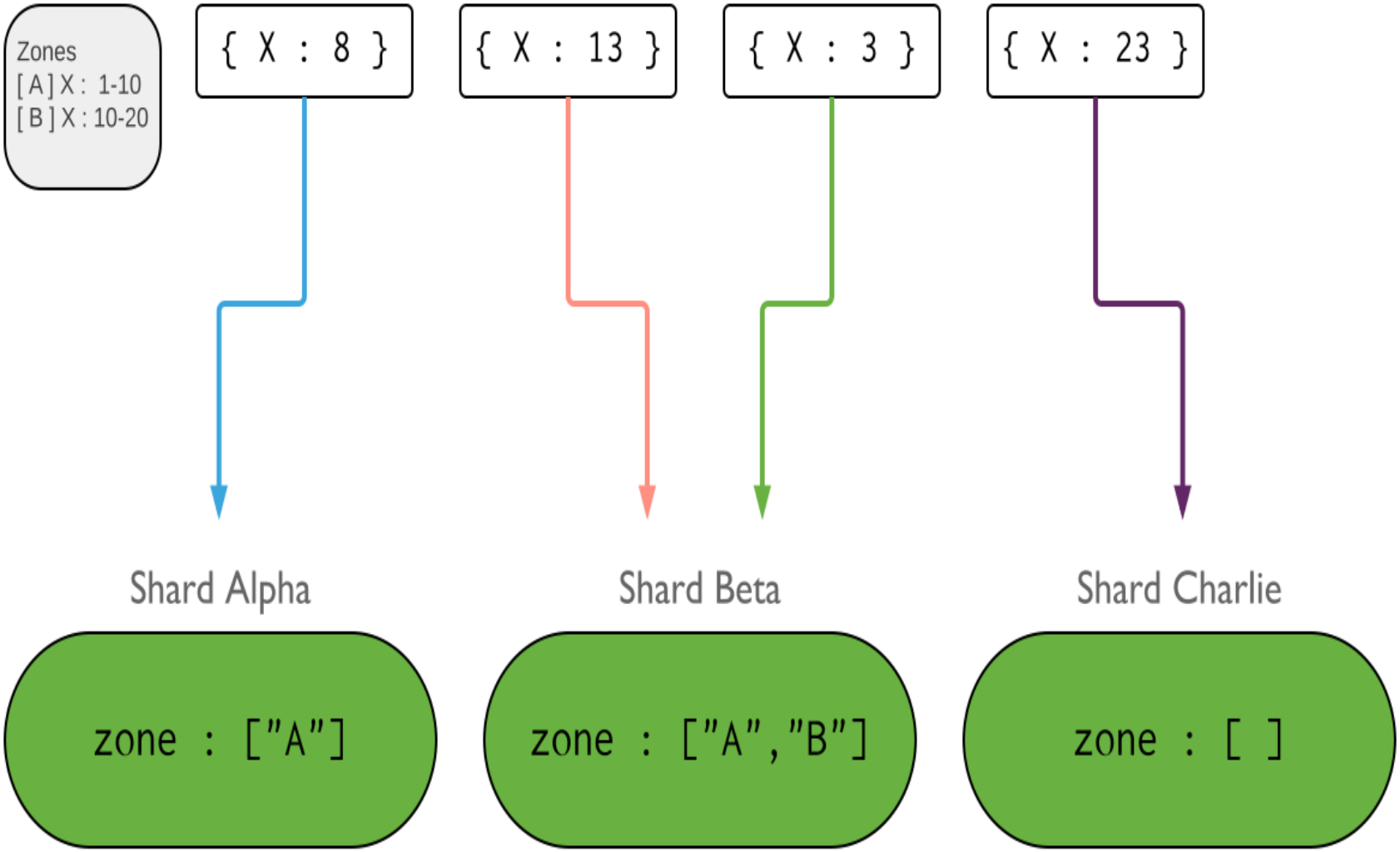
- **Ranged Sharding**
  - Involves dividing data into ranges based on the shard key values.
  - Each chunk is then assigned a range based on the shard key values
  - Allow targeted operations, but poorly chosen shard key may result in uneven data distribution



Sharding in MongoDB

- **Zoned Sharding**
  - Zones can help improve the locality of data for sharded clusters that span multiple data centers.
  - In sharded clusters, you can create zones of sharded data based on the shard key.
  - You can associate each zone with one or more shards in the cluster. A shard can associate with any number of zones.
  - In a balanced cluster, MongoDB migrates chunks covered by a zone only to those shards associated with the zone.
  - Each zone covers one or more ranges of shard key values

Zoned Sharding



Choosing a Shard Key

- The shard key is either an indexed field or indexed compound fields that determines the distribution of the collection's documents among the cluster's shards.
- All sharded collections must have an index that supports the shard key; i.e. the index can be an index on the shard key or a compound index where the shard key is a prefix of the index
- The choice of shard key affects the creation and distribution of the chunks across the available shards. This affects the overall efficiency and performance of operations within the sharded cluster.