Virtualization Techniques
Topics Today

Motivation
Virtual Machines
Containers (next lecture)
Motivation

● Suppose Srinidhi has a machine with 4 CPUs and 64GB of memory, and three customers:
  ○ Yuvraj wants a machine with 1 CPU and 24GB of memory
  ○ Dave wants 2 CPUs and 8GB of memory
  ○ Daniel wants 1 CPU and 32GB of memory
● What should Srinidhi do?
Motivation

- Srini can sell each customer a **virtual machine** (VM) with the requested resources
  - From each customer’s perspective, it appears as if they had a physical machine all by themselves (**isolation**)

![Diagram showing virtual machines and a physical machine connected through a virtual machine monitor.](image-url)
Virtualization

• “a technique for hiding the physical characteristics of computing resources from the way in which other systems, applications, or end users interact with those resources. This includes making a **single physical resource appear to function as multiple logical resources**; or it can include making **multiple physical resources appear as a single logical resource**”

Adapted from: Ken Birman
The history: from 1960’s

- IBM VM/370 – A VMM for IBM mainframe
  - Multiple OS environments on expensive hardware
  - Desirable when few machine around
- Popular research idea in 1960s and 1970s
  - Entire conferences on virtual machine monitors
  - Hardware/VMM/OS designed together
  - Allowed multiple users to share a batch oriented system
- Interest *died out* in the 1980s and 1990s
  - Hardware got more cheaper
  - Operating systems got more powerful (e.g. multi-user)

Adapted from: Ken Birman
Return of Virtual Machines

● Disco: Stanford research project (SOSP ’97)
  ○ Run commodity OSes on scalable multiprocessors
  ○ Focus on high-end: NUMA, MIPS, IRIX

● Commercial virtual machines for x86 architecture
  ○ VMware Workstation (1999-)
  ○ Connectix VirtualPC (now Microsoft)

● Research virtual machines for x86 architecture
  ○ Xen (SOSP ’03)
    ■ Xen and the art of virtualization - cited 8000+

● OS-level virtualization
  ○ FreeBSD Jails, Linux namespace

Adapted from: Ken Birman
Virtualization and Distributed Systems

- Running applications on their own virtual machines which include the related libraries and OS.
  - No need to worry about the diversity of platforms.
- Easy to manage large scale distributed systems (e.g., CDN) by virtualizing the whole environment.
  - E.g., migrate a server from one site to another.
Types of Virtualization

- System virtualization
  - Virtualizing the entire hardware and software layers

- Process virtualization
  - Virtualizing OS resources between processes
  - Language-level: Java, .NET, Smalltalk
  - OS-level: Solaris Zones, BSD Jails, Linux namespace (e.g., Docker, LXC) (Later in today’s lecture)
Types of Virtualization

Examples

- VM
- Guest OS
- Hypervisor (Type 2)
- Host OS
- Server

- Container
- Host OS
- Server

- App A
- App A'
- App B

- Bins/Libs
Topics Today

Motivation
Virtual Machines
Containers
Starting Point: A Physical Machine

- Physical Hardware
  - Processors, memory, chipset, I/O devices, etc.
  - Resources often grossly underutilized

- Software
  - Tightly coupled to physical hardware
  - Single active OS instance
  - OS controls hardware
A Virtual Machine

- **Software Abstraction**
  - Behaves like hardware
  - Encapsulates all OS and application state

- **Virtualization Layer**
  - Extra level of *indirection*
  - Decouples hardware, OS
  - Enforces isolation
  - Multiplexes physical hardware across VMs

Adapted from: Eyal DeLara
Types of System Virtualization

● Type 1: Native/Bare metal
  ○ Higher performance
  ○ VMWare ESX, Xen, Hyper-V

● Type 2: Hosted
  ○ Easier to install
  ○ Leverage host’s device drivers
  ○ VMware Workstation, Parallels
Virtual Machine Monitor (Hypervisor)

- **Classic Definition (Popek and Goldberg ’74)**
  
  A virtual machine is ... an efficient, isolated duplicate of the real machine. ... the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

- **VMM Properties**
  - **Fidelity**: Programs running in the virtualized environment run identically to running natively.
  - **Performance**: A statistically dominant subset of the instructions must be executed directly on the CPU.
  - **Safety and isolation**: A VMM must completely control access to system resources.
Requirements of Virtualization

● Isolation
  ○ Fault isolation
  ○ Performance isolation (+ software isolation, …)

● Encapsulation
  ○ Cleanly capture all VM state
  ○ Enables VM snapshots, clones

● Portability
  ○ Independent of physical hardware
  ○ Enables migration of live, running VMs (freeze, suspend,…)
  ○ Clone VMs easily, make copies

● Interposition
  ○ Transformations on instructions, memory, I/O
  ○ Enables transparent resource overcommitment, encryption, compression, replication …
Implementation
VMM Implementation Goals

● Should efficiently virtualize the hardware
  ○ Provide illusion of multiple machines
  ○ Retain control of the physical machine

● Which subsystems should be virtualized?
  ○ Processor ⇒ Processor Virtualization
  ○ Memory ⇒ Memory Virtualization
  ○ I/O Devices ⇒ I/O virtualization
Processor Virtualization

- An architecture is **classically(strictly virtualizable)** if all its sensitive instructions (those that violate safety and encapsulation) are a subset of the privileged instructions.
  - All instructions either privileged or non-privileged.
    - Privileged instruction: trap
    - Non-privileged instruction: execute natively
CPU Virtualization: Trap-and-Emulate

- **Privileged instructions** (e.g., Update CPU state, Manipulate page table): Trap to VMM
- **Non-privileged instructions** (e.g., Load from mem): Run as native machine

What if some instructions are in both (depending on CPU state)?
CPU Virtualization: x86 virtualization

- Special instructions: behaves differently depending on the CPU state (kernel or user)
  - x86 has 17+ special instructions
- VMs always run in user mode!
  - No trap by special instructions
- Techniques to address inability to virtualize x86
  - Replace non-virtualizable instructions with easily Virtualized ones statically (Para-virtualization)
  - Perform Binary Translation (Full Virtualization)
  - HW supports “guest” mode (Intel VT-x, AMD-V)
Virtualizing the CPU

● The VMM still need to multiplex VMs on CPUs
● How could this be done?
  ○ # Physical CPUs more than #Virtual CPUs?
  ○ # Virtual CPUs more than #Physical CPUs?
● Timeslice the VMs, each VM runs OS/Apps
● Use simple CPU scheduler
  ○ Round robin, work-conserving (give extra to other VM)
  ○ Can oversubscribe and give more #VCPUs that actual
Memory Virtualization

- OS assumes that it has full control over memory
  - Management: Assumes it owns it all
  - Mapping: Assumes it can map any Virtual $\rightarrow$ Physical

However, VMM partitions memory among VMs
  - VMM needs to assign hardware pages to VMs
  - VMM needs to control mapping for isolation
    - Cannot allow OS to map any Virtual $\Rightarrow$ hardware page

Adapted from: Alex Snoeren
Virtualization: Three Abstractions of Memory

- **Logical**: process address space in a VM
- **Physical**: abstraction of hardware memory. Managed by guest OS
- **Machine**: actual hardware memory (e.g. 2GB of DRAM)
x86 Memory Management Primer

- The processor operates with virtual addresses
- Physical memory operates with physical addresses
- x86 includes a hardware translation lookaside buffer (TLB)
  - Maps virtual to physical page addresses
- x86 handles TLB misses in HW
  - HW walks the page tables ⇒ Inserts virtual to physical mapping

![Diagram of memory management process]
Shadow page table

- VMM creates and manages page tables that map **virtual pages** directly to **machine pages**
- VMM needs to keep its V \(\Rightarrow\) M tables consistent with changes made by OS to its V \(\Rightarrow\) P tables
  - VMM maps OS page tables as read only
  - When OS writes to page tables, trap to VMM
  - VMM applies write to shadow table and OS table, returns
  - Also known as memory tracing
Shadow Page Tables (cont')

- VMM creates and manages page tables that map virtual pages directly to machine pages
  - These tables are loaded into the MMU on a context switch
  - VMM page tables are the shadow page tables
- VMM needs to keep its V ⇒ M tables consistent with changes made by OS to its V⇒P tables
  - VMM maps OS page tables as read only
  - When OS writes to page tables, trap to VMM
  - VMM applies write to shadow table and OS table, returns
  - Also known as memory tracing
  - Again, more overhead...

Adapted from: Alex Snoeren
Memory Management

● VMMs tend to have simple memory management
  ○ Static policy: VM gets 8GB at start
  ○ Dynamic adjustment is hard since OS cannot handle
  ○ No swapping to disk

● More sophistication: Overcommit with ballooning
  ○ Balloon driver runs inside OS ⇒ consume hardware pages
  ○ Balloon grows or shrinks (gives back mem to other VMs)

● Even more sophistication: memory de-duplication
  ○ Identify pages that are shared across VMs!
Memory Ballooning

inflated balloon (+ pressure)

may page out to virtual disk

guest OS manages memory implicit cooperation

deflated balloon (− pressure)

may page in from virtual disk
I/O Virtualization

- Direct access: VMs can directly access to devices
  - Requires H/W support (e.g., DMA passthrough, SR-IOV)
- Shared access: VMM provides an emulated device and routes I/O data to and from the device and VMs
Virtualizing I/O Devices

- However, overall I/O is complicated for VMMs
  - Many short paths for I/O in OSes for performance
  - Better if hypervisor needs to do less for I/O for guests
  - Possibilities include direct device access, DMA pass-through
    - need H/W support!

- Networking also complex as VMM and guests all need network access
  - VMM can bridge guest to network (direct access)
  - VMM can provide network address translation (NAT)
  - NAT address local to machine on which guest is running, VMM provides address translation to guest to hide its address
VM Storage Management

● VMM provides “virtual disks”
  ○ Type 1 VMM – store guest root disks and config information within file system provided by VMM as a disk image
  ○ Type 2 VMM – store the same info as files in the host OS’ file system

● Example of supported operations:
  ○ Duplicate file (clone) → create new guest
  ○ Move file to another system → move guest
  ○ Convert formats: Physical-to-virtual (P-to-V) and Virtual-to-physical (V-to-P)
  ○ VMM also usually provides access to network attached storage (just networking) ⇒ live migration
Live migration

- Running guest OS can be moved between systems, without interrupting user access to the guest or its apps
- Supported by type 1 hypervisors
- Very useful for resource management, no downtime, etc
- How does it work?

1. Source VMM connects to the target VMM
2. Target VMM creates a new guest (e.g. create a new VCPU, etc)
3. Source sends all read-only guest memory pages to the target
4. Source sends all RD/WR pages to the target, marking them clean
5. Source repeats step 4, as some pages may be modified ⇒ dirty
6. When cycle of steps 4 and 5 becomes very short, source VMM freezes guest, sends VCPU’s final state, sends final dirty pages, and tells target to start running the guest
7. Target acknowledges that guest running ⇒ source terminates guest
Live migration

0 – Running 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

6 – Running Guest Target

7 – Terminate Guest Source
Virtual Machine Summary

- VMMs multiplex virtual machines on hardware
  - Virtualize CPU, Memory, and I/O devices
  - Run OSes in VMs, apps in OSes unmodified
  - Run different versions, kinds of OSes simultaneously

- Support for virtualization built into Intel and AMD CPUs
  - Goal is to fully virtualize architecture
  - Transparent trap-and-emulate approach now feasible

- System virtualization seems to be heavy
  - Long boot time, clone, resume entire system …
  - Is there any lighter virtualization method which provides enough isolation? ⇒ Container!
Types of Virtualization

- Full virtualization (e.g. VMWare ESX)
  - Unmodified OS, virtualization is transparent to OS
  - VM looks exactly like a physical machine
- Para virtualization (e.g. XEN)
  - OS modified to be virtualized,
  - Better performance at cost of transparency

Attribution
http://forums.techarena.in/guides-tutorials/1104460.htm
Topics Today

Motivation
Virtual Machines
Containers

Motivation for Containers
Implementation in Linux
Practical Implications
Containers in a Distributed System
Motivation for Containers

Architecture of web applications is changing

Classical architecture

Monolithic application
100 engineers
Release / month
Horizontal scale out

Components
- Login
- Personification
- Renderer
- Ads
- Suggestions
- Encoders

Potential limitations of this architecture?

- .WAR too big for IDE?
- Async release of updates?
- Change tech of a component?
- Failure Isolation?
Motivation for Containers

Changing architecture of web applications

**New architecture:** components → “micro services”

Define API between components
10-20 engineers / component
Components release and scale independently

Components
- Login
- Personification
- Renderer
- Ads
- Suggestions
- Encoders

Potential limitations of this new architecture?
- Per-component overhead?
- How to define services?
- API/Communication latency?
Prominent Example: Netflix

Migration to micro services: 2008-2016
Hundreds of services, complex dependencies

Source: Netflix Blog
https://medium.com/netflix-techblog/a-microscope-on-microservices-923b906103f4
Micro Services vs Virtual Machines

Overheads associated with deploying on VMs

- I/O overhead
- OS-startup overhead per VM
- Memory/Disk overhead (duplicate data)

Perception: too much overhead for micro services

Still want virtualization properties → Containers
Overview: Container Virtualization

Idea:

- Multiple isolated instances of programs
- Running in user-space (shared kernel)
- Instances see only resources (files, devices) assigned to their container

Requires Operation System support

Many other names:

OS-level virtualization, partitions, jails (FreeBSD jail or chroot jail)

It’s a container if “I can ssh into it and it feels like a VM.”
Requirements for Container Virtualization

-**Isolation and encapsulation**
  - Fault and performance isolation
  - Encapsulation of environment, libraries, etc.

-**Low overhead**
  - Fast instantiation / startup
  - Small per-operation overhead (I/O, ..)

- (Portability)
- (Security Isolation)
- Interposition (no hypervisor)
Implementing Container Virtualization

Key problems:
- Isolating which resources containers see
- Isolating resource usage (security & performance)
- Efficient per-container filesystems

The container ecosystem
Orchestrating containers in a distributed system
Limitations of containers
Resource View Isolation

Problem: containers should only see “their” resources, and are the only
users of their resource
(e.g., process IDs (PIDs), hostnames, users IDs (UIDs), interprocess communication (IPC))

Solution: each process is assigned a “namespace”
- Syscalls only show resources within own namespace
- Subprocesses inherit namespace

Current implementation: namespace implementation per resource type
(PIDs, UIDs, networks, IPC), in Linux since 2006

Practical implication:
- Containers feel like VMs, can get root
- Security relies on kernel, containers make direct syscalls
Resource Usage Isolation

Problem: meter resource usage and enforce hard limits per container
(e.g., limit memory usage, priorities for CPU and I/O usage)

Solution: usage counters for groups of processes (cgroups)
- Compressible resources (CPU, I/O bandwidth): rate limiting
- Non-compressible resources (Memory/disk space): require terminating containers (e.g., OOM killer per cgroup)

Current implementation: cgroups/kernfs, in Linux since 2013/2014

Practical implication:
- Efficiency: 1000s of containers on a single host
- Small overhead per memory allocation, and in CPU scheduler
Problem: per-container filesystems without overhead of a “virtual disk” for each container

Solution: layering of filesystems (copy on write):
- Read-write (“upper”) layer that keeps per-container file changes
- Read-only (“lower”) layer for original files

Current implementation: OverlayFS, in Linux since 2014

Practical implication:
- Instant container startup
- “Upper” layer is ephemeral

<table>
<thead>
<tr>
<th>Upper</th>
<th>/index.html</th>
<th>/photo/cat.jpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower</td>
<td>/index.html</td>
<td></td>
</tr>
</tbody>
</table>
Advantages of Container Virtualization

Fast boot times:
100s of milliseconds
(10s-100s of seconds for VMs)

High density:
1000s of containers per machine

Very small I/O overhead
The Container Ecosystem

Docker (also: LXC, Google Lmctfy)

Libcontainer (written in GO)
- Automates using kernel features (namespaces, cgroups, OverlayFS)
- Container-image configuration language

FROM golang
WORKDIR /go/src
COPY ./src .
RUN go-wrapper install monitor
CMD ./start.sh
## Container Virtualization Variants

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Operating system</th>
<th>Year</th>
<th>FS isolation</th>
<th>Copy on Write</th>
<th>Disk quota</th>
<th>I/O limits</th>
<th>Mem limits</th>
<th>CPU quota</th>
<th>Network isolation</th>
<th>Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>chroot</td>
<td>most UNIX-like operating systems</td>
<td>1982</td>
<td>Partial[a]</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Docker</td>
<td>Linux,[7] FreeBSD,[8] Windows, macOS</td>
<td>2013</td>
<td>Yes</td>
<td>Yes</td>
<td>Not directly</td>
<td>Yes (since 1.10)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (since 1.10)</td>
<td>Yes (since 1.10)</td>
</tr>
<tr>
<td>Linux-VServer</td>
<td>Linux, Windows Server</td>
<td>2001</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial[c]</td>
<td>Partial[d]</td>
<td></td>
</tr>
<tr>
<td>OpenVZ</td>
<td>Linux</td>
<td>2005</td>
<td>Yes</td>
<td>Yes (ZFS)</td>
<td>Yes</td>
<td>Yes[8]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes[11]</td>
</tr>
<tr>
<td>FreeBSD jail</td>
<td>BSDs</td>
<td>2000[22]</td>
<td>Yes</td>
<td>Yes (ZFS)</td>
<td>Yes[23]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes[27]</td>
</tr>
</tbody>
</table>
Containers are Widely Used

“The container has become the sole runnable entity supported by the Google infrastructure.”

All Google services run on containers
  (gmail, search, etc)

Support offered by all IAAS providers
  (GCD container engine, Azure container service, AWS ECS)

Large scale orchestration (micro services)
  (Kubernetes, Docker compose/swarm, Apache Mesos)

The Container Orchestration Challenge

How to launch 2+ billion containers / week?

- Scheduling: where should my containers run?
- Naming and discovery: where are my containers now?
- Aggregation: compose sets of containers (with dependencies)
- Monitoring: what is happening inside my containers?
- Fault-tolerance: when to terminate / restart containers?
- Scaling: when and how to make compositions smaller/bigger?
- Deployment: which code version, how to rollback to prev version?
Container Orchestration Tools

Key element: consistent view of system state

System state

User API

Desired state

Scheduler

System state in replicated data store

Services: group containers into hierarchies

Service discovery: name services + overlay networks
Many Open Problems

Container scheduling at scale
- How many containers can be packed?
- When to move them?
- How wide should a service be spread?
  (Recall ML-speedup curves - Google’s scheduler assumes linear speedup)

Automating configuration and failure recovery
- Predicting your service’s resource usage is hard
- Knowing when to replicate/terminate services is also hard
Limitations of Containers

No isolation guarantee for some resources

- Resources that are not managed by the kernel
  - Last-level processor caches
  - Memory bandwidth

Large attack surface under adversarial behavior

- Containers typically have access to all syscalls
  - Linux offers 400 syscalls (10 new syscalls/year)
- One approach: syscall filtering (very complicated)
- In cloud environments: additional level of isolation (VMs)
Summary: Virtual Machines and Containers

VMs

Strengths: strong isolation guarantees
Weaknesses: OS startup, disk, memory, and hypervisor overhead

Containers

Strength: fast startup times, negligible I/O overheads, very high density
Weaknesses: weak security isolation

In practice: techniques complement each other

Use VMs to isolate between different users, and containers to isolate different applications/services of a single user
Extra Material
Other Types of Virtualization
Language Level Virtualization

- not-really-virtualization but using same techniques, providing similar features
  - Programming language is designed to run within custom-built virtualized environment (e.g. Oracle JVM)
- 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
  - Programs written in Java run in the JVM no matter the underlying system
Types of VMs – Emulation

- Emulation allows guests to run on different CPU (VMs require guest to be recompiled for each new CPU)
  - (e.g., company replaces outdated servers with new servers containing different CPU architecture, but still want to run old applications)

- Translate all guest instructions from guest CPU to native CPU
  - (Thus: emulation, not virtualization)

- Performance challenge – order of magnitude slower than native code
  - (New machines faster than older machines, which reduces slowdown)

- Where do you think it is used still?

- Very popular – especially in gaming where old consoles emulated on new
VMM Example: XEN
Example VMM: XEN : Introduction

• A Para-Virtualized Interface
• Can host multiple and different OSes
• Supports Isolation
• Performance Overhead is minimum
• Can Host up to 100 Virtual Machines

• Trivia: started at Cambridge, sold for a lot of $$
• Open Source software, xen.org at the moment
XEN: Approach

• Drawbacks of Full Virtualization with respect to x86 architecture
  • Support for virtualization not inherent in x86 architecture
  • Certain privileged instructions did not trap to the VMM
  • Virtualizing the MMU efficiently was difficult
  • Other than x86 architecture deficiencies, it is sometimes required to view the real and virtual resources from the guest OS point of view

• Xen’s Answer to the Full Virtualization problem:
  • It presents a virtual machine abstraction that is similar but not identical to the underlying hardware -para-virtualization
  • Requires Modifications to the Guest Operating System
Terminology Used

- **Guest Operating System (OS)** – refers to one of the operating systems that can be hosted by XEN.
- **Domain** – refers to a VM within which a Guest OS runs with applications on top of the OS.
- **Hypervisor** – XEN (VMM) itself.
- **Guest OS’s (domU) and priviledge domain “dom0”**
XEN’s VMI : Memory Management

• Problems
  • x86 architecture uses a hardware managed TLB
  • Segmentation

• Solutions
  • One way would be to have a tagged TLB, which is currently supported by some RISC architectures
  • Guest OS are held responsible for allocating and managing the hardware page tables but under the control of Hypervisor
  • XEN should exist (64 MB) on top of every address space

• Benefits
  • Safety and Isolation
  • Performance Overhead is minimized

Adapted from: Ken Birman
XEN’s VMI : CPU

• Problems
  • Inserting the Hypervisor below the Guest OS means that the Hypervisor will be the most privileged entity in the whole setup
  • If the Hypervisor is the most privileged entity then the Guest OS has to be modified to execute in a lower privilege level
  • Exceptions

• Solutions
  • x86 supports 4 distinct privilege levels – rings
  • Ring 0 is the most and Ring 3 is the least
  • Allowing the guest OS to execute in ring 1- provides a way to catch the privileged instructions of the guest OS at the Hypervisor
  • Exceptions such as memory faults and software traps are solved by registering the handlers with the Hypervisor
  • Guest OS must register a fast handler for system calls with the Hypervisor
  • Each guest OS will have their own timer interface

Adapted from: Ken Birman
XEN’s VMI: Device I/O

• Existing hardware Devices are not emulated
• A simple set of device abstractions are used – to ensure protection and isolation
• Data is transferred to and fro using shared memory, asynchronous buffer descriptor rings – performance is better
• Hardware interrupts are notified via a event delivery mechanism to the respective domains
XEN : Cost of Porting Guest OS

- Linux is completely portable on the Hypervisor - the OS is called XenoLinux

- Lot of modifications to the architecture specific code was done in both the Oses

- In comparing both OSes – Larger Porting effort for XP

Adapted from: Ken Birman
XEN : Control and Management

- Xen exercises just basic control operations such as access control, CPU scheduling between domains etc.

- All the policy and control decisions with respect to Xen are undertaken by management software running on one of the domains – domain0

- The software supports creation and deletion of
XEN: Detailed Design

- Control Transfer
  - Hypercalls – Synchronous calls made from domain to XEN
  - Events – Events are used by Xen to notify the domain in an asynchronous manner

- Data Transfer
  - Transfer is done using I/O rings
  - Memory for device I/O is provided by the respective domain
  - Minimize the amount of work to demultiplex data to a specific domain
XEN: CPU Scheduling

- Xen uses Borrowed Virtual Time scheduling algorithm for scheduling the domains.
- Per domain scheduling parameters can be adjusted using domain0.

Advantages:

- Work – Conserving
- Low – Latency Dispatch by using virtual time warping
XEN : Time and Timers

• Guest OSes are provided information about real time, virtual time and wall clock time

• **Real Time** – Time since machine boot and is accurately maintained with respect to the processor’s cycle counter and is expressed in nanoseconds

• **Virtual Time** – This time is increased only when the domain is executing – to ensure correct time slicing between application processes on its domain

• **Wall clock Time** – an offset that can be added to the current real time.
XEN : Disk Management

• Only Domain0 has direct unchecked access to the physical disks.
• Other Domains access the physical disks through virtual block devices (VBDs) which is maintained by domain0.
  • VBS comprises a list of associated ownership and access control information, and is accessed via I/O ring.
• A translation table is maintained for each VBD by the hypervisor, the entries in the VBD’s are controlled by domain0.
• Xen services batches of requests from competing domains in a simple round-robin fashion.