Emulators and Virtualization Platforms



ILLINOIS

illinois.edu

x86 Virtualization

- X86 operating systems are designed to run directly on the bare-metal hardware
- Virtualizing the x86 architecture requires placing a virtualization layer under Ring 0
- some sensitive instructions can't effectively be virtualized as they have different semantics when they are not executed in Ring 0
- trapping and translating these sensitive and privileged instruction requests at runtime was the challenge that originally made x86 architecture virtualization look impossible

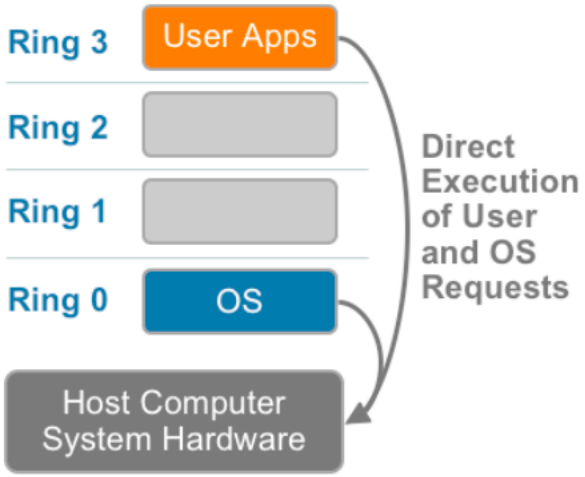


Figure: x86 privilege level architecture without virtualization

Virtualization techniques

- three techniques exist for handling sensitive and privileged instructions to virtualize the CPU on the x86 architecture
  - Full virtualization using binary translation
  - OS assisted virtualization or paravirtualization
  - Hardware assisted virtualization

Full Virtualization

- Refers to running an unmodified OS on a virtual machine
- virtualize any x86 operating system using a combination of binary translation and direct execution techniques
- translates kernel code to replace nonvirtualizable instructions with new sequences of instructions that have the intended effect on the virtual hardware
- user level code is directly executed on the processor
- Full virtualization is the only option that requires no hardware assist or operating system assist to virtualize sensitive and privileged instructions.

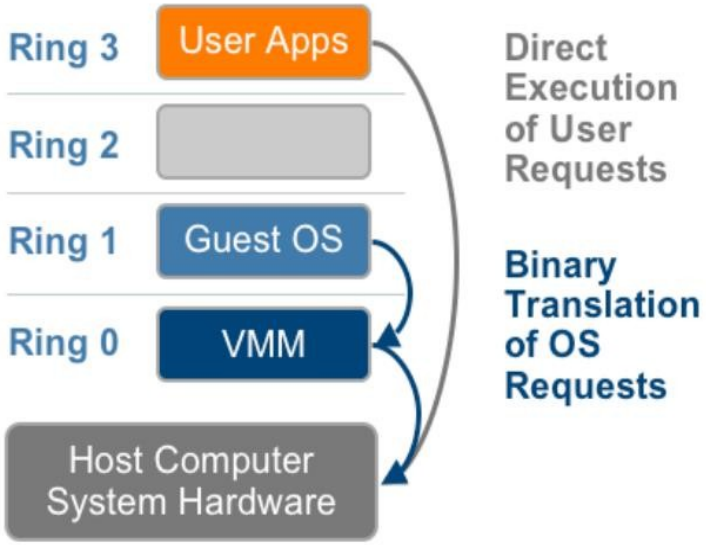


Figure: The Binary translation approach to x86 virtualization

Paravirtualization

- OS Assisted Virtualization
- Communication between guest OS and hypervisor to improve performance and efficiency
- Modifying the OS: replace nonvirtualizable instructions with hypercalls
- Hypervisor provides hypercall interfaces for other critical kernel operations such as memory management, interrupt handling and time keeping.
- If the source code for the OS is not available ?

Paravirtualization

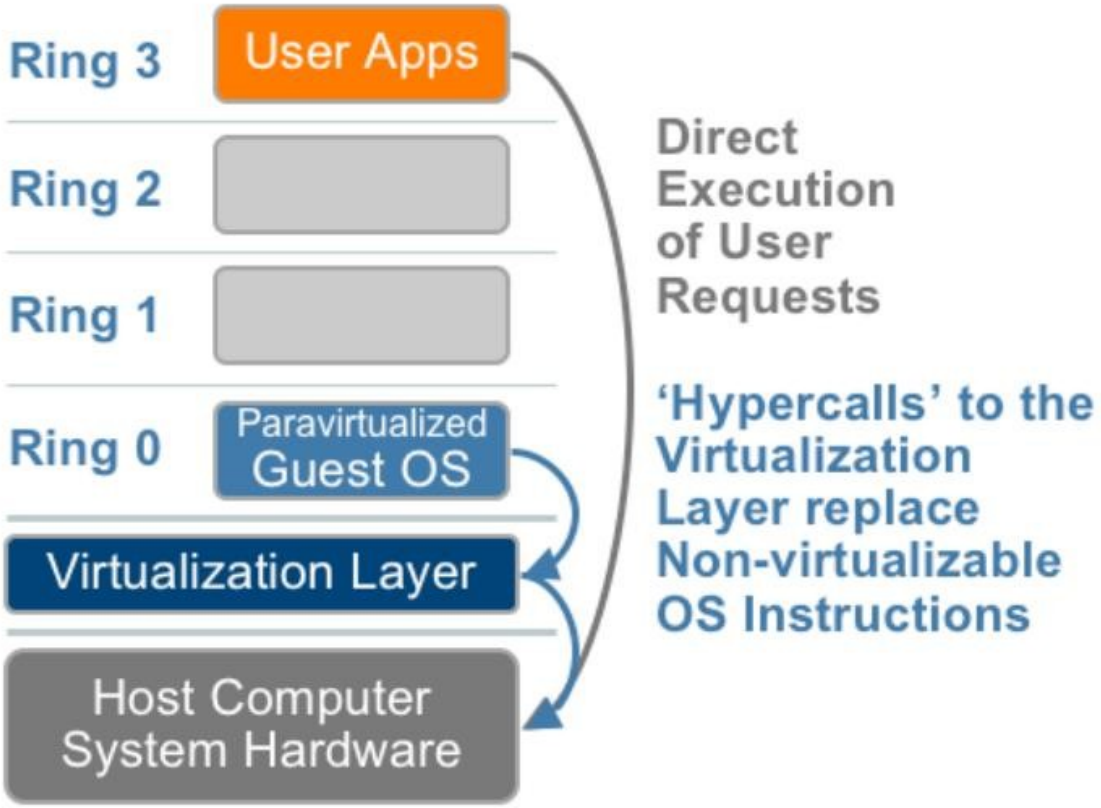


Figure: The Paravirtualization approach to x86 Virtualization

Hardware-assisted Virtualization

- Intel Virtualization Technology (VT) Xeon, AMD-Virtualization (AMD-V) Opteron
- new CPU execution mode feature that allows the VMM to run in a new root mode below ring 0
- privileged and sensitive calls are set to automatically trap to the hypervisor
- no need for paravirtualization or binary translation
- Processors with Intel VT and AMD-V became available in 2006

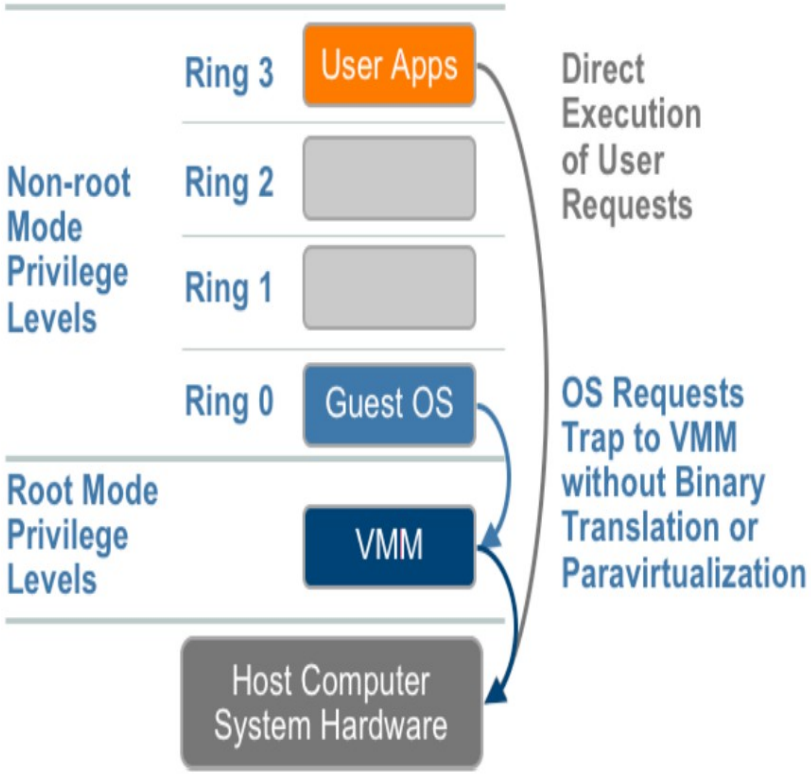
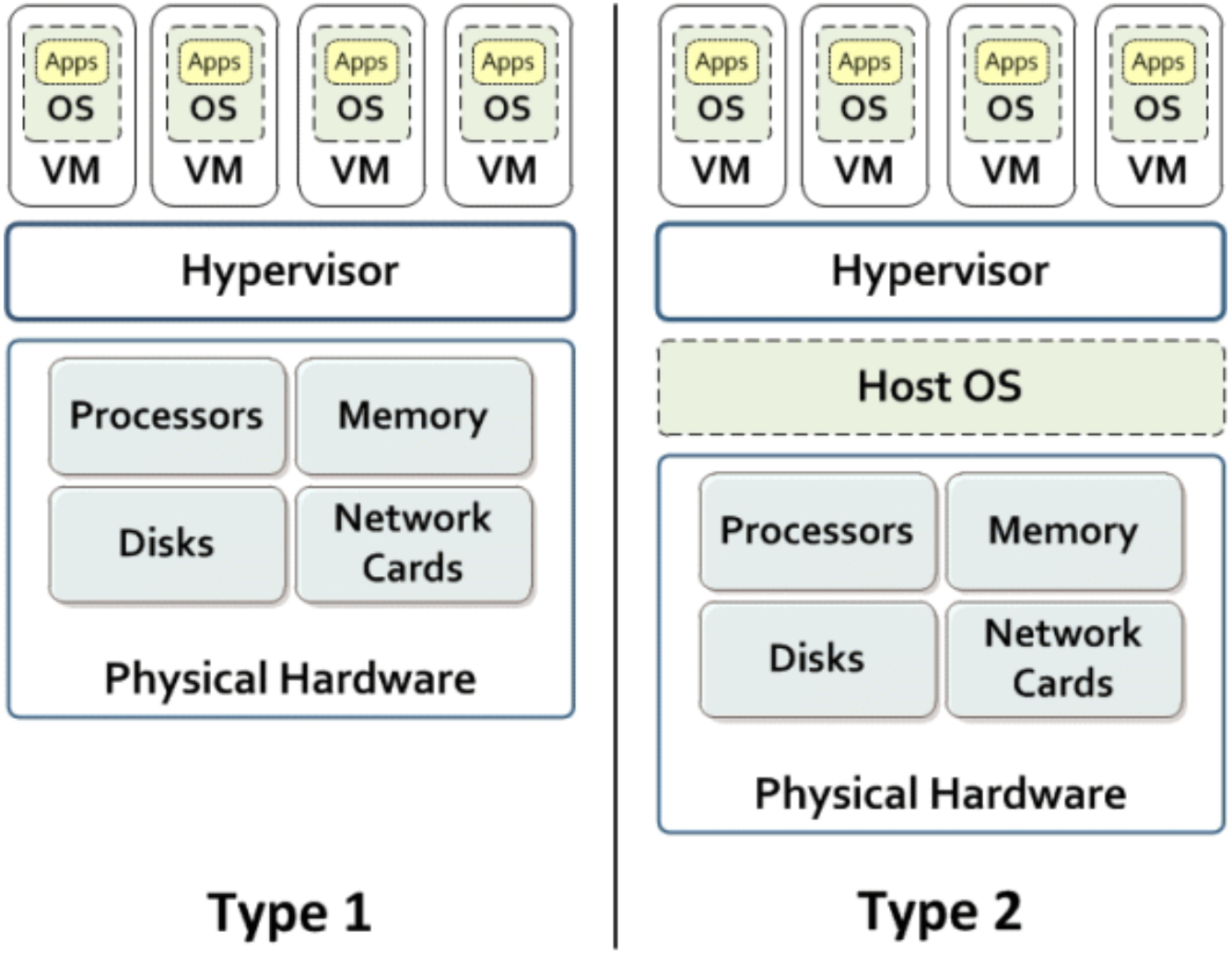


Figure: The hardware assist approach to x86 virtualization

Types of Hypervisors

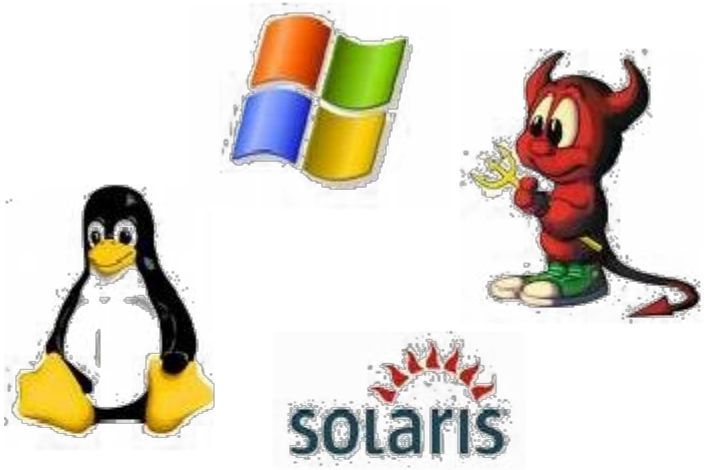



Xen

Xen

- A layer of software that runs directly on the computer hardware replacing the Operating System
- Allows running of multiple guest operating systems concurrently
- Supports x86, x86-64, Itanium, PowerPC, and ARM architectures
- Supports Linux, NetBSD, FreeBSD, Solaris, Windows, and more

Xen Hypervisor





Hardware

Hardware

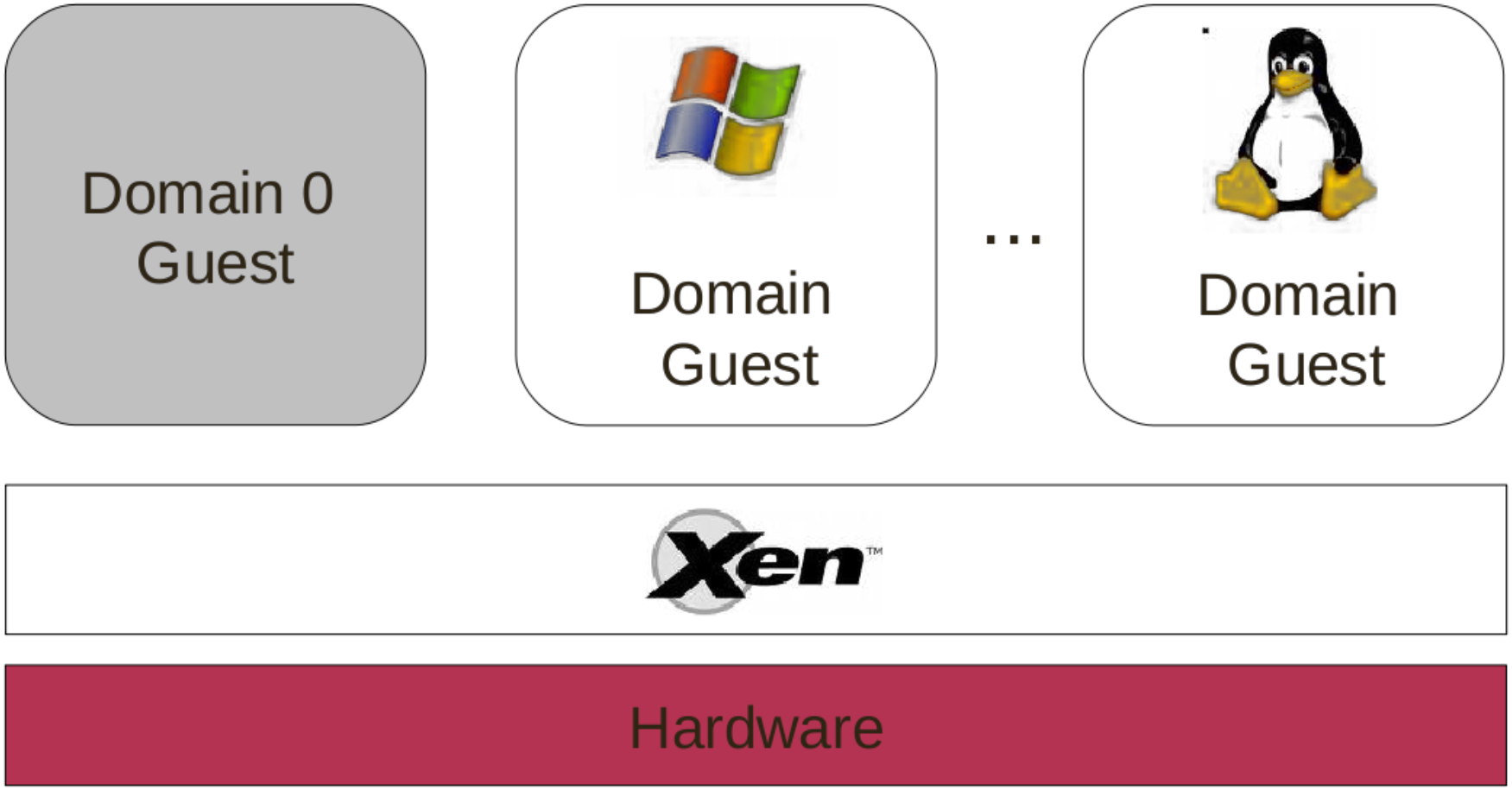
Standard Computer
Hardware + OS

Virt. Computer
Hardware + Xen

Three Components

- Xen Hypervisor
- Domain 0 (Dom 0) : Privileged guest running on hypervisor with direct hardware access and guest management responsibilities
- Multiple DomainU (DomU) : Unprivileged guests running on hypervisor without direct access to hardware

Components



Xen Hypervisor

- Launching the most privileged domain ("dom0")

- Memory management
  - Allocating memory to guest OS
  - Scrubbing free memory (ECC)
  - Protecting guest OS from each other
  - Providing translation services between address spaces

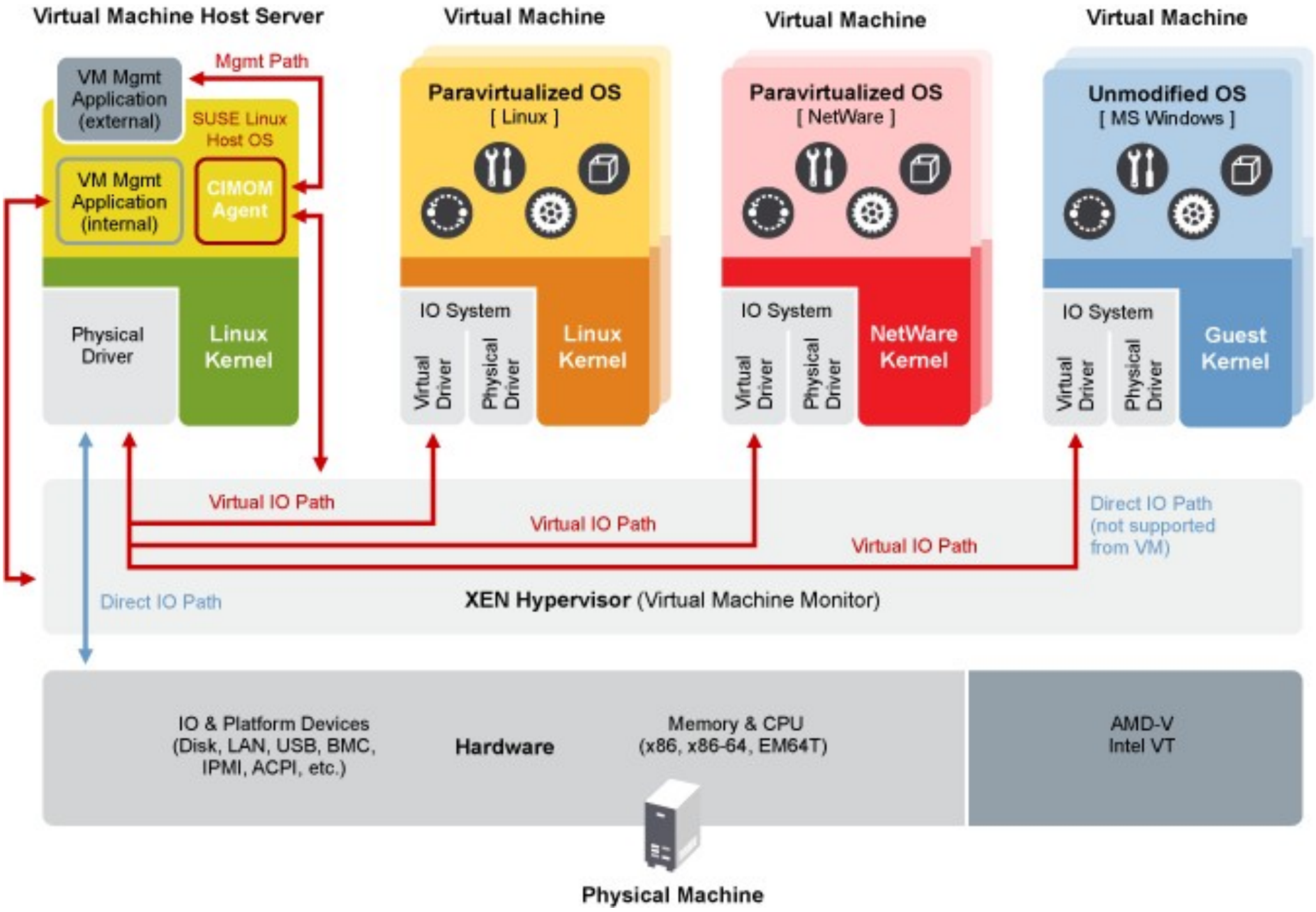
- CPU scheduling for all virtual machines (domains)
  - **Borrowed Virtual Time (BVT) -** dispatches the runnable VM with the smallest virtual time first
  - **Simple Earliest Deadline First (SEDF) -** uses real-time algorithms to deliver guarantees.
  - **Credit Scheduler -** features automatic load balancing of virtual CPUs across physical CPUs on an SMP host

Dom0

- The sysadmin profile
- Launched during initial system start-up
- Can run any OS other than Windows
- Has unique privileges
- Can manage starting, stopping, I/O requests of DomUs
- Dom0 has drivers for hardware, and it provides Xen virtual disks and network access for guests

DomU

- Launched by Dom0
- Paravirtualization
  - OS is aware that it is running on hypervisor
  - Requires modified guest OS
- Hardware Virtual Machine
  - OS unaware of running on hypervisor
  - Requires special hardware


Virtual Machine Host Server
Virtual Machine
Virtual Machine
Virtual Machine
Mgmt Path
VM Mgmt Application (external)
SUSE Linux Host OS
VM Mgmt Application (internal)
CIMOM Agent
Physical Driver
Linux Kernel
Paravirtualized OS [Linux]
IO System
Virtual Driver
Physical Driver
Linux Kernel
Paravirtualized OS [NetWare]
IO System
Virtual Driver
Physical Driver
NetWare Kernel
Unmodified OS [MS Windows]
IO System
Virtual Driver
Physical Driver
Guest Kernel
Virtual IO Path
Virtual IO Path
Virtual IO Path
Direct IO Path (not supported from VM)
Direct IO Path
XEN Hypervisor (Virtual Machine Monitor)
IO & Platform Devices (Disk, LAN, USB, BMC, IPMI, ACPI, etc.)
Hardware
Memory & CPU (x86, x86-64, EM64T)
AMD-V Intel VT
Physical Machine

Dalvik: process virtual machine



http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf. Very good pdf about DVM can be downloaded using this link.

Dalvik: process virtual machine

- Every Android application runs in its own process, with its own instance of the DVM.

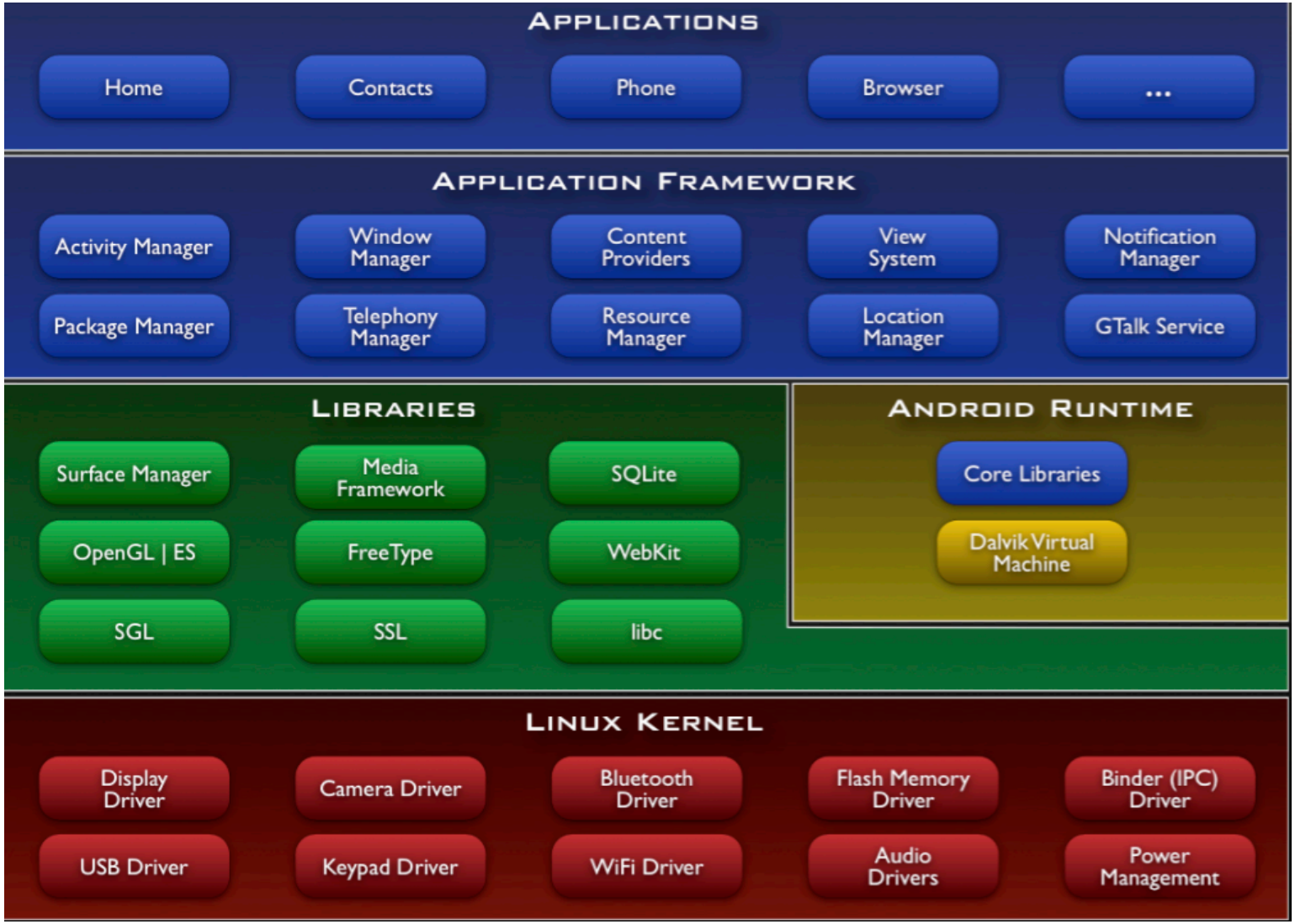
- The DVM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

Motivation

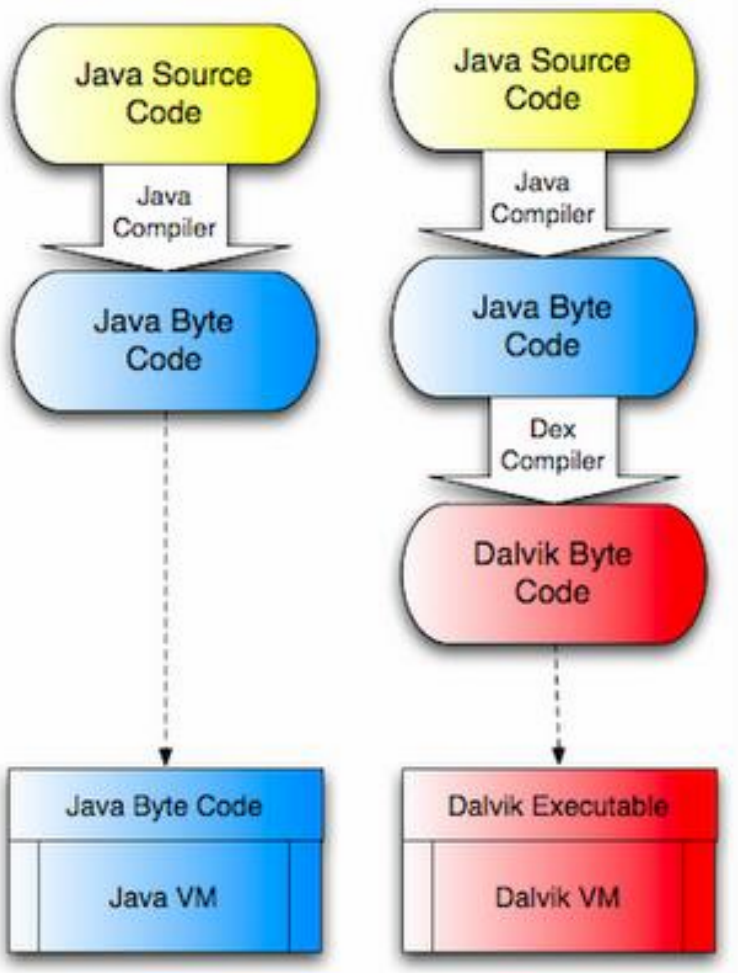
- Given the extremely wide range of target environments on Android devices, it is critical for the application platform to be abstracted away from the underlying operating system and hardware device.

- In summary, the Android runtime must support the following:
  - limited processor speed
  - limited RAM
  - no swap space
  - battery powered
  - diverse set of devices
  - sandboxed application runtime

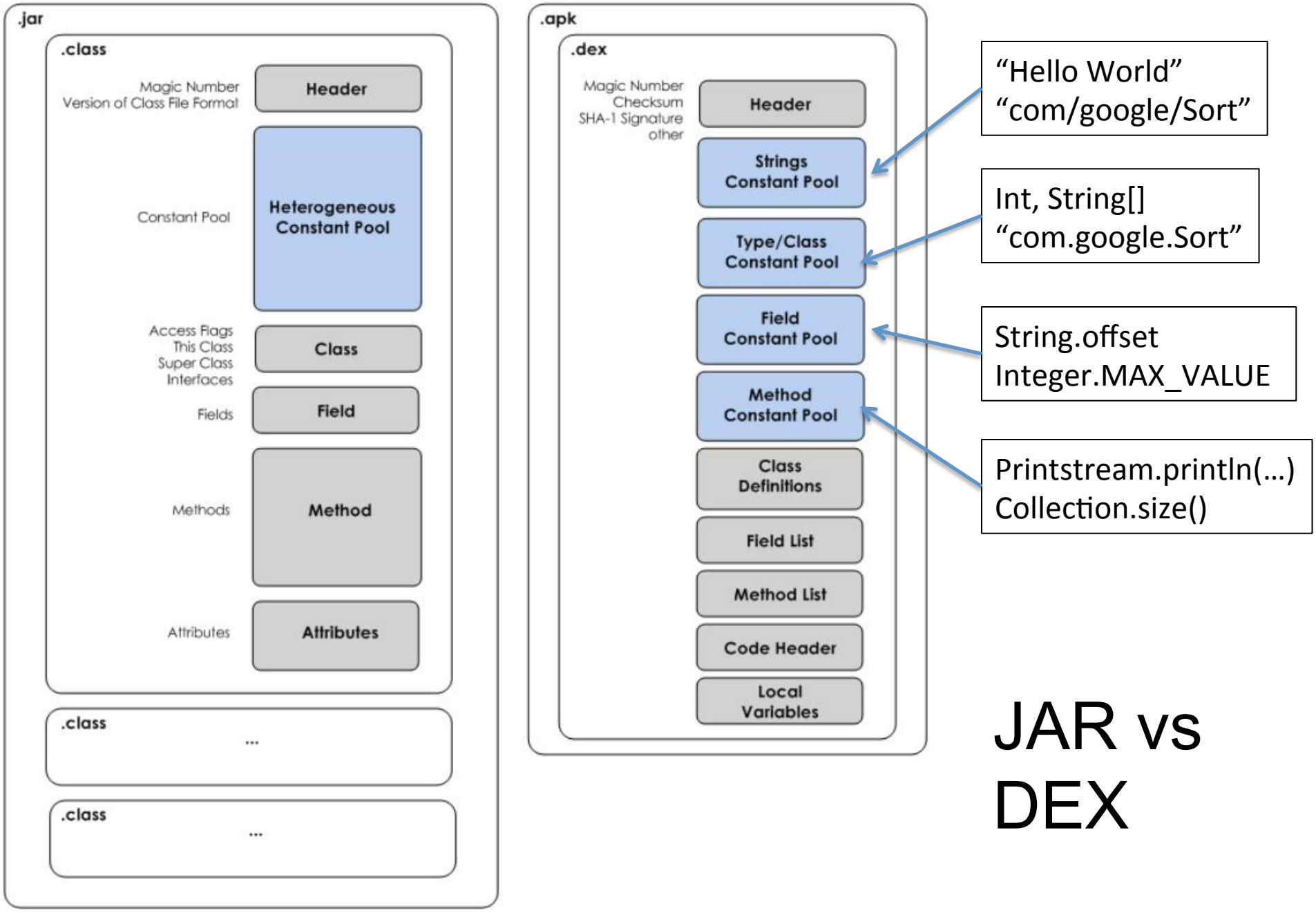
Environment structure



Dalvik: process virtual machine



The compact Dalvik Executable format(.dex) is designed to be suitable for systems that are constrained in terms of memory and processor speed.



JAR vs DEX

http://sites.google.com/site/io/dalvik-vm-internals. Very good PPT about DVM can be downloaded using this link.

JAR vs DEX size comparison

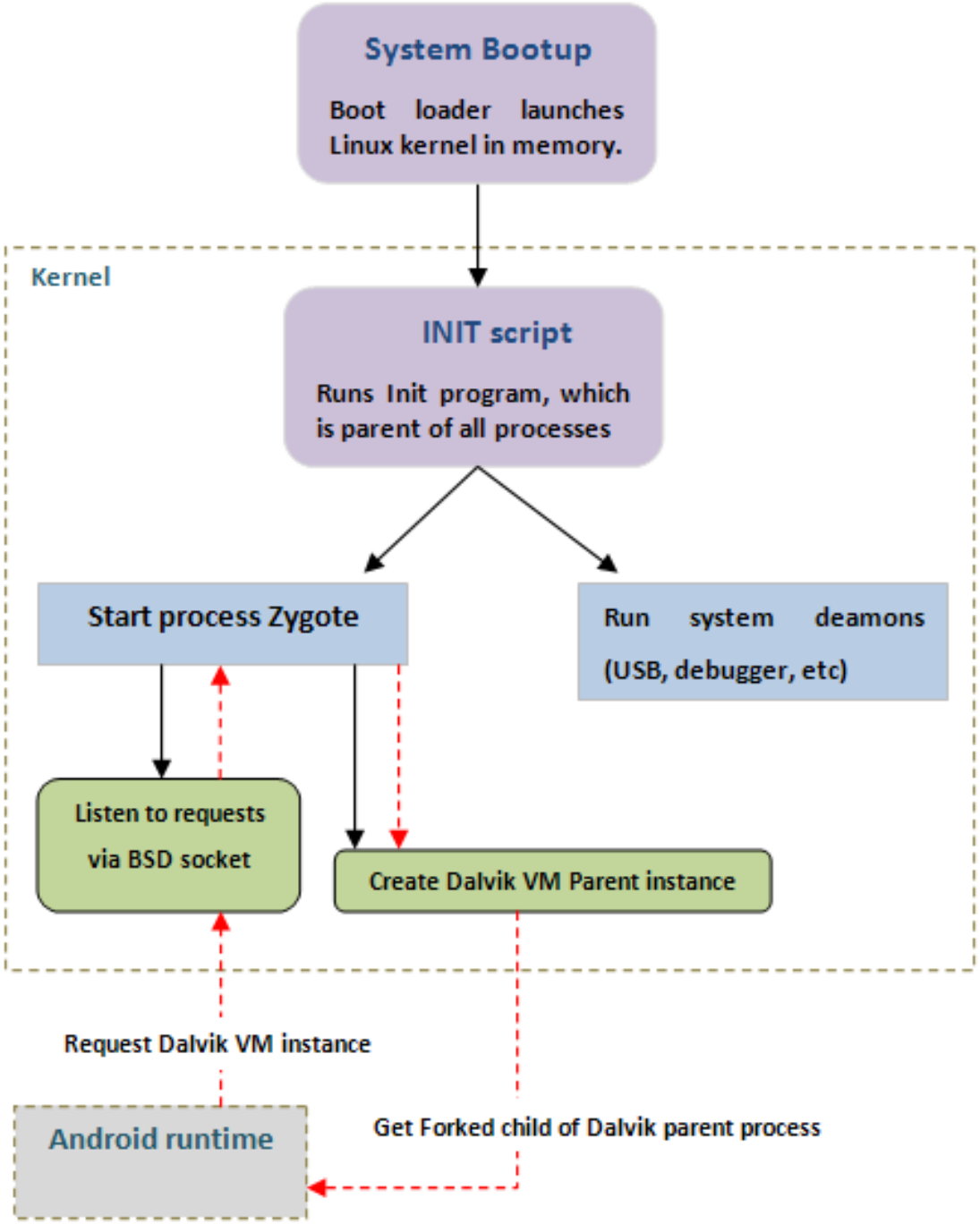
Code	Uncompressed JAR File (bytes)	Compressed JAR File (bytes)	Uncompressed dex File (bytes)
Common System Libraries	21,445,320 (100%)	10,662,048 (50%)	10,311,972 (48%)
Web Browser App	470,312 (100%)	232,065 (49%)	209,248 (44%)
Alarm Clock App	119,200 (100%)	61,658 (52%)	53,020 (44%)

The Zygote: Motivation

- Android uses a concept called the Zygote to enable both sharing of code across VM instances and to provide fast startup time of new VM instances.
- The Zygote design assumes that there are a significant number of core library classes and corresponding heap structures that are used across many applications.
- These are the data and classes that most applications use but never modify.
- These characteristics are exploited to optimize sharing of this memory across processes.

The Zygote: Design

- The Zygote is a VM process that starts at system boot time.
- It initializes a Dalvik VM, which preloads and preinitializes core library classes.
- Generally these core library classes are read-only and are therefore a good candidate for preloading and sharing across processes.
- Once the Zygote has initialized, it will sit and wait for socket requests coming from the runtime process indicating that it should fork new VM instances based on the Zygote VM instance.



Android OS boots up

Allowing writes to classes

- The core library classes are generally only read, but not written to by applications.

- When those classes are written to, the memory from the shared Zygote process is copied to the forked child process of the application's VM and written to there.

- *Copy-on-Write* behaviour allows for maximum sharing of memory while still prohibiting applications from interfering with each other and providing security across application and process boundaries

QEMU: Quick EMUlator



Qemu

QEMU

- Machine emulator + Virtualizer

- Modes:
  - User-mode emulation: allows a (Linux) process build for one CPU to be executed on another
    - **QEMU as a “Process VM”**
  - System-mode emulation: allows emulation of a full system, including processor and assorted peripherals
    - **QEMU ad a “System VM”**

- Popular uses:
  - For cross-compilation development environments
  - Virtualization
  - Android Emulation

Dynamic Binary Translation

- Works like a JIT compiler, but doesn't include an interpreter
- All guest code undergoes binary translation
  - Guest code is split into "translation blocks”
  - A translation block is similar to a basic block in that the block is always executed as a whole (ie. no jumps in the middle of a block).
- Translation blocks are translated into a single sequence of host instructions and cached into a translation cache.
  - Cached blocks are indexed using their guest virtual address (ie. PC count), so they can be found easily
  - Translation cache size can vary (32 MB by default)
  - Once the cache runs out of space, the whole cache is purged

TCG – Tiny code generator

- Modern QEMU (version 0.10.0 and newer) uses a proper intermediate representation to do translation
- Guest instructions are translated into TCG ops by the front end (target-∗/translate.c)
- TCG ops are then translated into host CPU instructions by the back end (tcg/∗/tcg-target.c).
- The TCG instruction set is very primitive, making it difficult to implement complex instructions using it. For this reason, TCG supports ”helper functions”
- Helper pseudo-ops can be emitted that will be compiled into native function calls. This way, complex instructions can be implemented in C

Block Chaining

- Normally, the execution of every translation block is surrounded by the execution of special code blocks called the prologue and the epilogue
- The prologue initializes the processor for generated host code execution and jumps to the code block. The epilogue restores normal state and returns to the main loop
- Returning to the main loop after each block adds significant overhead, which adds up quickly
- QEMU can patch the original block to jump directly into the next block instead of jumping to the epilogue

Other (major) QEMU components

- Memory address translation
  - Software-controlled MMU (model) to translate target virtual addresses to host virtual addresses
  - Mapping between Guest virtual address and host virtual addresses
  - Mapping between Guest virtual address and registered I/O functions for that device

- Device emulation
  - i440FX host PCI bridge, Cirrus CLGD 5446 PCI VGA card , PS/2 mouse & keyboard, PCI IDE interfaces (HDD, CDROM), PCI & ISA network adapters, Serial ports, PCI UHCI USB controller & virtual USB hub

References

- https://www.vmware.com/files/pdf/VMware_paravirtualization.pdf
- http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf
- http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf
- https://www.youtube.com/watch?v=ptjedOZEXPM
- https://wiki.aalto.fi/download/attachments/41747647/qemu.pdf
- http://www.csd.uoc.gr/~hy428/reading/qemu-internals-slides-may6-2014.pdf