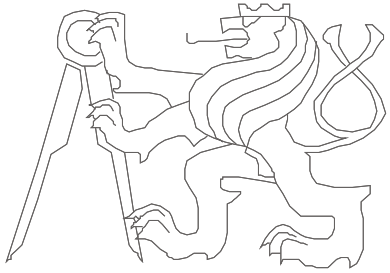
Computer Architectures

Multi-level computer organization, virtual machines



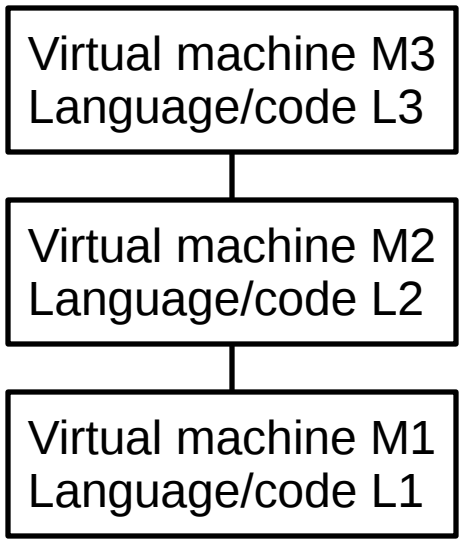
Czech Technical University in Prague, Faculty of Electrical Engineering

Multi-level computer organization

Machine code (language) - executed directly by CPU, instruction set
– level L1 – alphabet {0,1} , hard for humans, architecture specific
Higher-level languages – more user/programmer friendly, L2 + more

L2 program execution on machine supporting L1 language

Compilation - L2 instructions are replaced by sequences of L1 instructions
Interpretation – program codded in L1 performs according L2 accessed as data (slower)



L3 PRG. is interpreted by program running on M2 or M1 system or compiled to L2 or L1

L2 PRG. is interpreted by program running on M1 system or compiled to L1 language

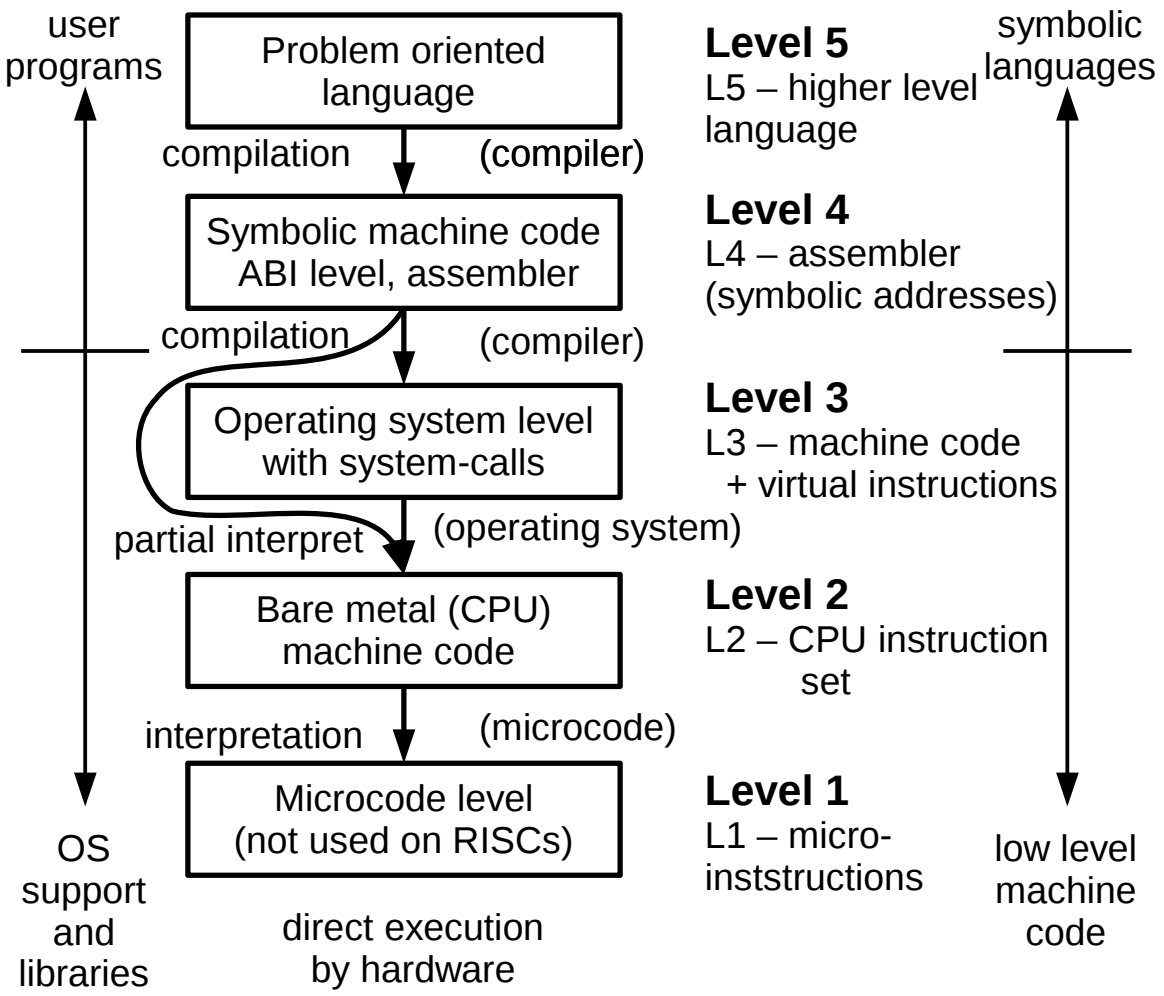
L1 PROGRAM is directly executed by computer hardware

Virtual machine concept

We declare virtual computer M_i for level i which executes language L_i.

Program written in language L_i is interpreted or translated for M_{i-1} computer, etc.

Todays multi level machine architecture



Machine levels evolution

- **the first computers – machine code only – 1 level in the stack**
- **50-ties - Wilkes – microcode – 2 levels**
- **60- ties – OS – 3 levels**
- **compilers, prog. languages – 4 levels**
- **user oriented applications - 5 levels**
- **HW and SW are logically equivalent (can be mutually substituted)**
- **competition and convergence of RISC and CISC CPU architectures and implementations**

Processes and their states

PROCESS – executed program (program – passive as data, process - active)

PROCESS STATE - information enough to continue previously frozen process

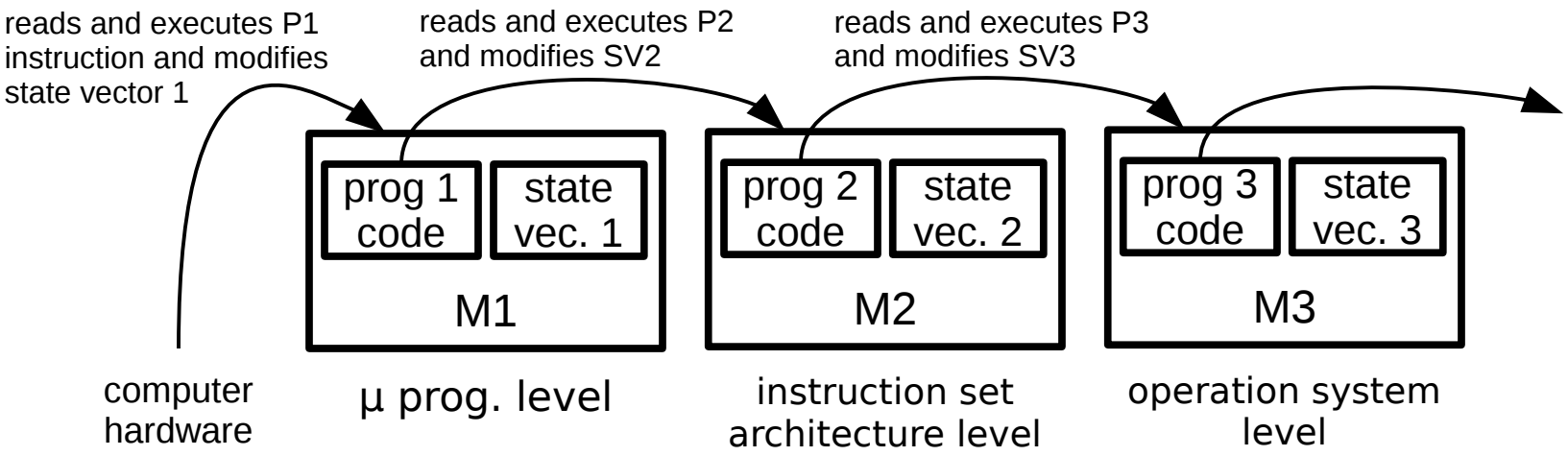
1. program
2. instruction pointer
3. variables values, data, registers
4. state and position for all I/O

ASUMPTION: process P does not modify its own program!

STATE VECTOR – variable components of process state – modified by HW or program

PROCESS = PROGRAM + STATE VECTOR

STATE VECTOR UPDATE – st. vector P2 is changes by P1 -> P1 is interpreter of P2 program



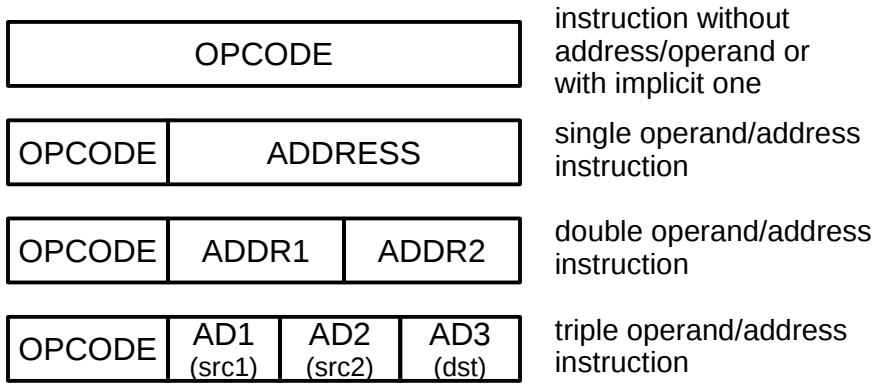
Machine Code – Instruction Set Architecture (M2)

Defined by instruction set (usually referred as processor architecture)
ISA – Instruction Set Architecture × Microarchitekture (implementation)

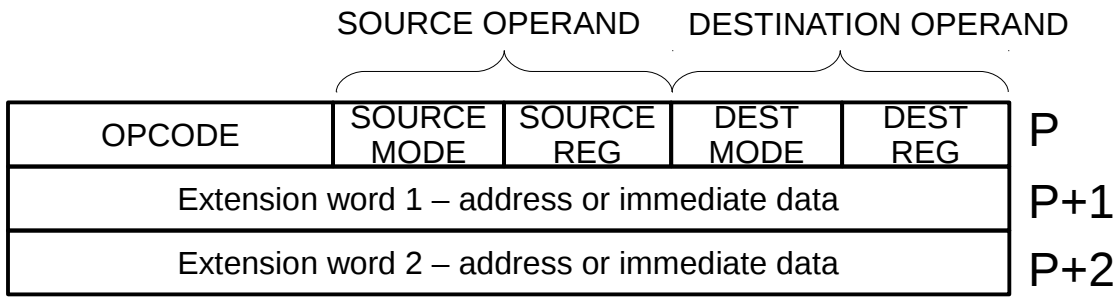
HW - structure of computer**, CPU, I/O, buses, memory organization, hardware latches, buffers, registers, clocks**
SW - instruction set**, data formats, addressing methods, CPU registers**

Instruction format

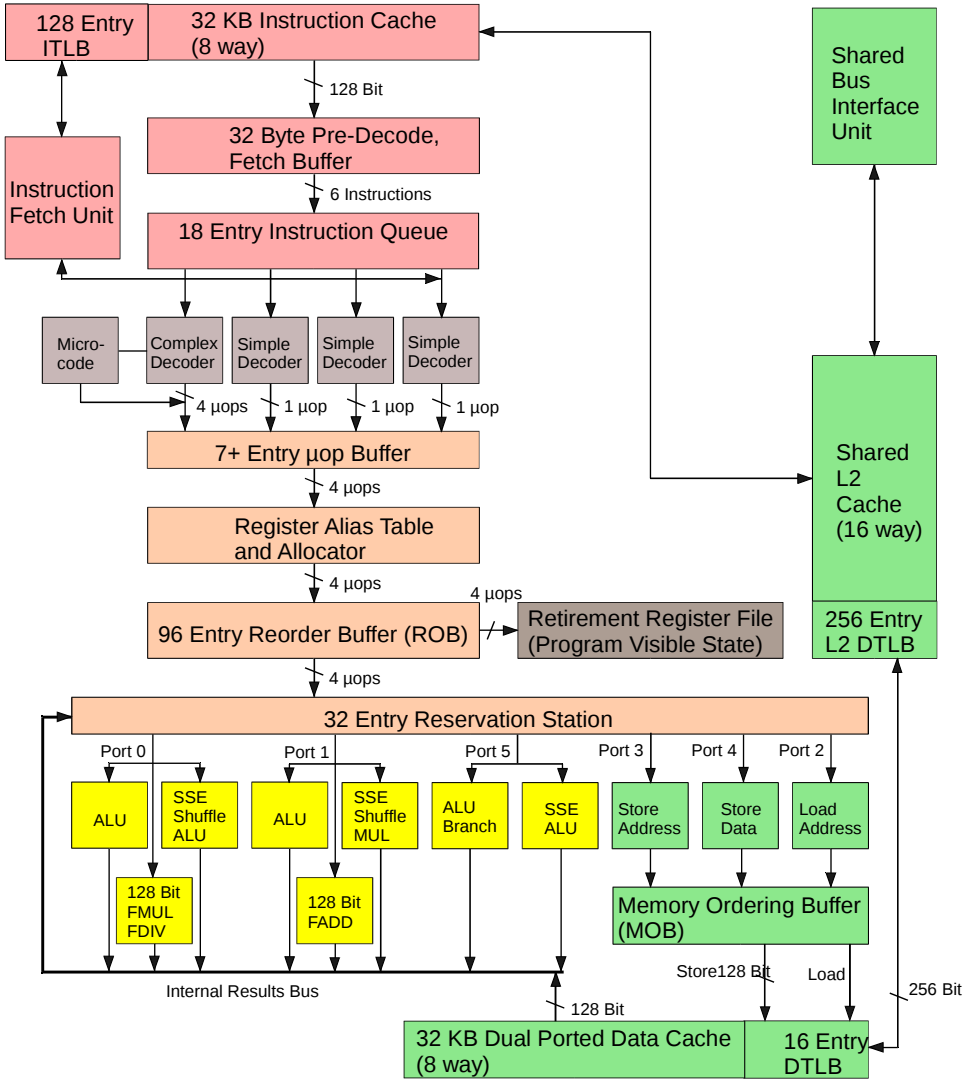
- one or more fields, (one or more continuous words)
- addr. fields contains address or specify register(s) (indirect)



Most commonly used CISC formats have two operands, RISC usually provide three operands but only register ones



Intel Core 2 Architecture (Mikr architecture) (M1)



Requirements for Good Programmer

Every programmer should be able to write a code that meets the following criteria:

- Efficient CPU utilization, ie. Fast code
- Efficient use of memory, ie. Use only such data types/structures that suffice for programmer purpose while consuming minimum/reasonable amount of memory space
- Follow laid down rules and convention, ie. Format source code for better reading and understanding (spaces, indentation, comments …)
- Made future modifications and improvement of code easy (using functions or OOP according to the environment and programmer skills)
- Solid and well-testable code, ie. Detect and correct possible errors (most critical and common are errors, which we do not expect that can occur)
- Documentation (instructions for using the program and future extension)

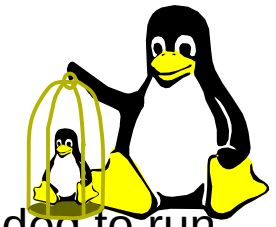
Loosely paraphrased/based on Randall Hyde book Write Great Code - Volume 2: Thinking Low-Level, Writing High-Level

⇒ my suggestions for x86 and other CPU architectures

- for x86 – read and understand Agner Fog's CPU reviews and optimization manuals http://www.agner.org/optimize/
  - The microarchitecture of Intel, AMD and VIA CPUs
  - An optimization guide for x86 platforms
- the great overview of CPU evolution and methods to speed up instructions execution
  - John Bayko: Great Microprocessors of the Past and Present
- understand how higher level languages and runtime environments translate language contsructs to underlaying data types, memory allocations and access pattern and use that as base to decide which data types select for given case
- study compiler properties and options and write benchmarks/tests and develop your "sense" for approximation about efficiency, price and memory demands of possible program constructs
- at the end you often find that many attempts to optimize code without additional knowledge and long term experience your clumsy optimization attempts can lead to slower code that code generated by compiler unconstrained ⇒ but if the performance is really a priority, application is demanding the there is only way to study and learn where initial knowledge is not sufficient, where to trust compiler, where it is necessary to help it (inline assembly, vektorization SSE etc.) or extend it
- all above effort is useless and waste of time if original algorithm or concept is wrongly selected
- see the example what are the ways to concatenate two bytes to form 16-bit word

Virtualization

- Virtualization hides the implementation/properties of lower layers (reality) and provides an environment with desired properties
- Virtualization can be divided to following techniques in computer technology
  - Purely application / language level and compiled code (byte-code) - virtual machines, for example. JVM, dotNET
  - Emulation and simulation typically different computer architecture (also called cross-virtualization)
  - Native virtualization - an isolated environment that provides the same type of architecture for the unmodified OS
  - Virtualization with full support in the HW
  - Partial virtualization - typically only address spaces
  - Paravirtualization - the operating system has to modified/extended to run in provided environment
  - OS-level virtualization - only separated user environments (jails, containers, etc.)



OS Level Virtualization

Mechanism	Operating system	License	Available	File system isolation	Copy on Write	Disk quotas
chroot	most UNIX-like operating systems	varies by operating system	1982	Partial[5]	No	
Docker	Linux[6]	Apache License 2.0	2013	Yes	Yes	Not directly
Linux-VServer (security context)	Linux	GNU GPLv2	2001	Yes	Yes	
lmctfy	Linux	Apache License 2.0	2013	Yes	Yes	
LXC	Linux	GNU GPLv2	2008	Yes[10]	Partial. Yes with Btrfs.	Partial. Yes with LVM
OpenVZ	Linux	GNU GPLv2	2005	Yes	No	
Virtuozzo	Linux, Windows	Proprietary	July 2000[14]	Yes	Yes	
Solaris Containers (Zones)	Solaris, OpenSolaris, Illumos	CDDL	February 2004	Yes	Yes (ZFS)	
FreeBSD jail	FreeBSD	BSD License	2000[20]	Yes	Yes (ZFS)	
sysjail	OpenBSD, NetBSD	BSD License	Last March 3, 2009	Yes	No	
WPARs	AIX	Proprietary	2007	Yes	No	
HP-UX Containers (SRP)	HPUX	Proprietary	2007	Yes	No	Partial. Yes with logical volumes
iCore Virtual Accounts	Windows XP	Proprietary/Freeware	2008	Yes	No	

Source: http://en.wikipedia.org/wiki/Operating-system-level_virtualization

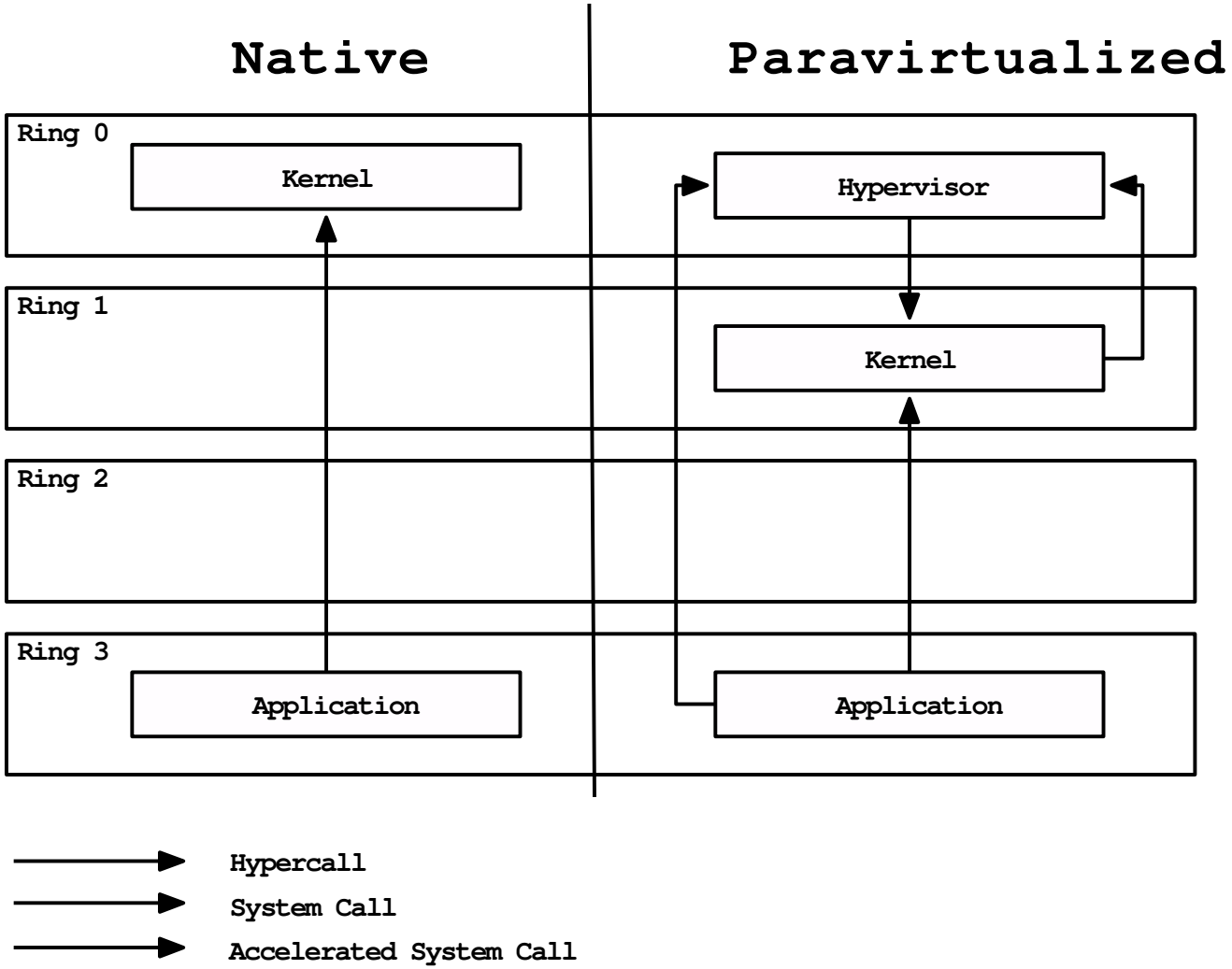
Full Computer System Virtualization

- Host system (often called domain DOM 0)
- Guest system
- Processor model implementation for guest system
  - for native cases – common code unprivileged instructions executed directly on the host CPU, privileged cause exception
  - for cross case – instructions are interpreted by emulator (probram in DOM 0), optionally accelerators and JIT techniques are used
- Attempt to execute privileged instructions in guest system
  - causes exception which is serviced by monitor/hypervisor by emulation effect on state vector guest system CPU or memory mapping
  - if CPU includes support for HW virtualization (AMD-V, Intel VT-x) then hardware can take care of such case, shadow pagetables atec.
- Peripherals/I/O devices
  - IO and memory mapped peripherals access attempt leads to exception and hypervisor emulates function and keeps state of such subsystems
  - guest system is adapted to pass I/O request (send packet, read disk block) directly in format which is understand by hypervisor (device drivers direcly supporting given hypervisor etc.)

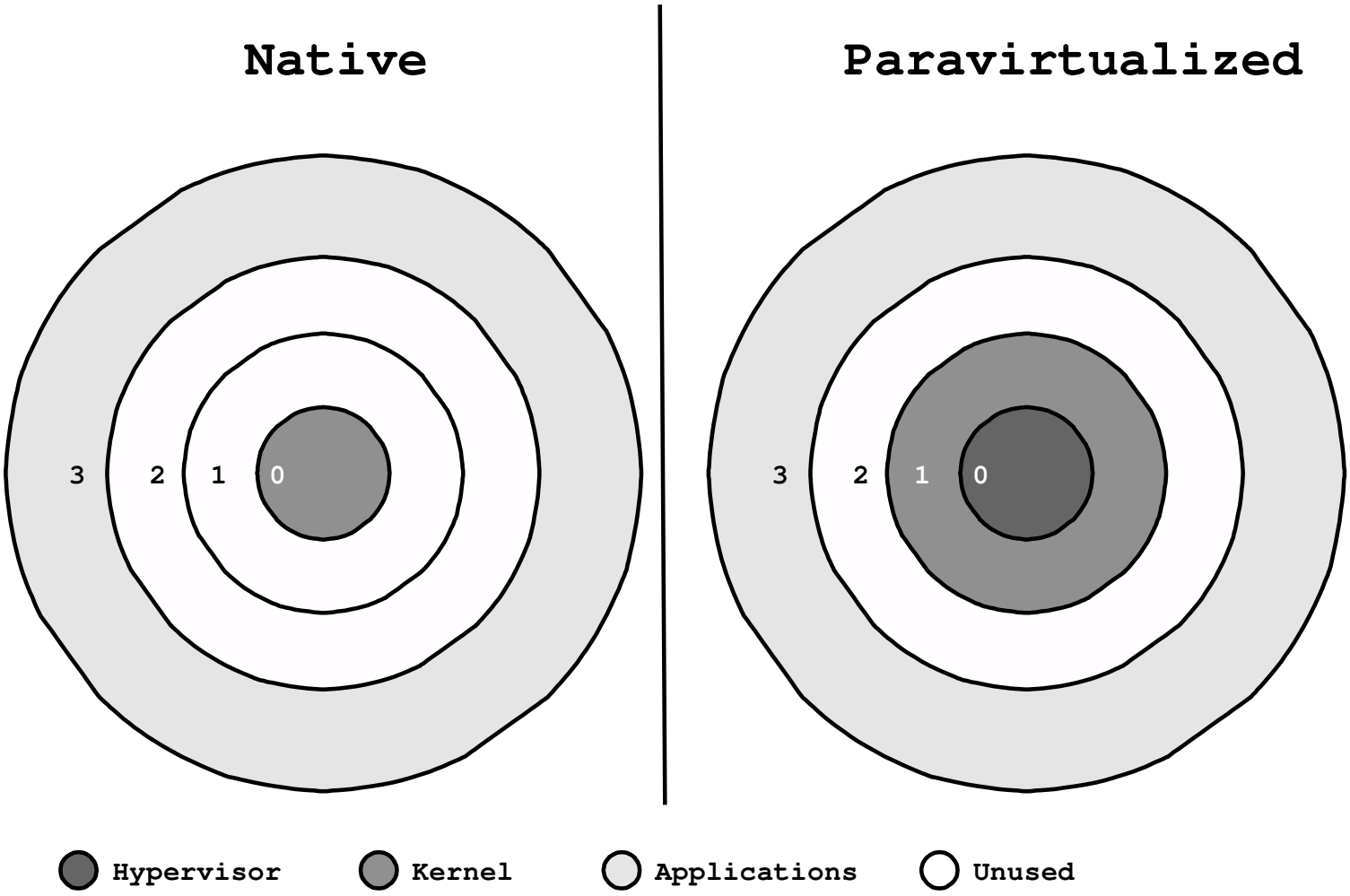
Hypervisor

- takes care to start and stop domains
- monitors their state end services exceptions – emulated effect of priviledged instructions
- divides/allocates memory and CPU time for individul guest systems
- emulates operations of peripheral devices and forwards data to/from device drivers API of physical devices and networks on host system level
- it can be implemented
  - in userspace of host system as unprivileged application (QEMU)
  - with HW support in host CPU and operating system (KVM)
  - as and independent system/microkernel which uses one other system in specialized domain (DOM 0) for communication with physical devices by providing direct HW access and transports data from this DOM 0 system to other (user DOM U) domains (XEN)

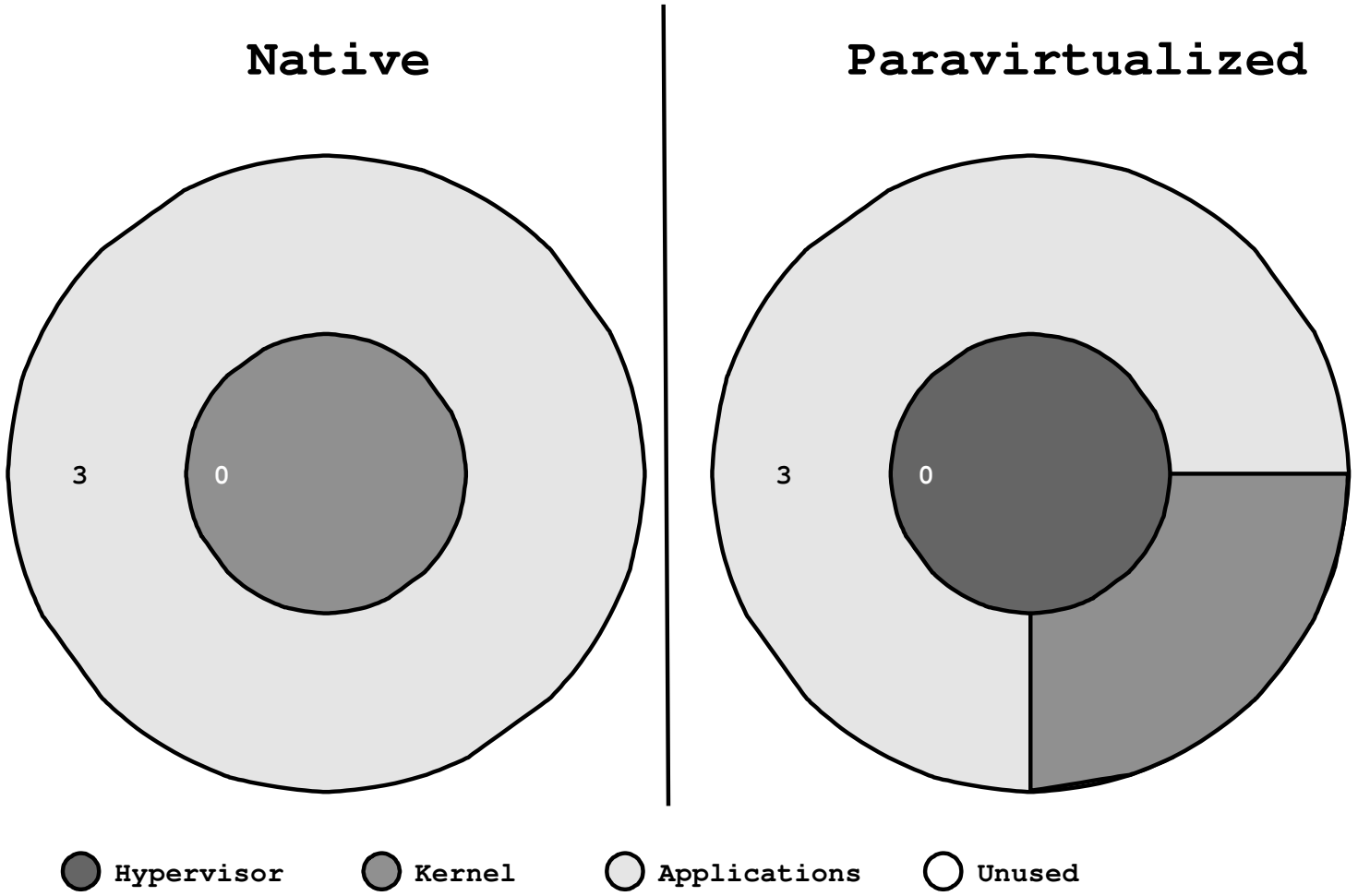
Paravirtualized system calls (XEN)



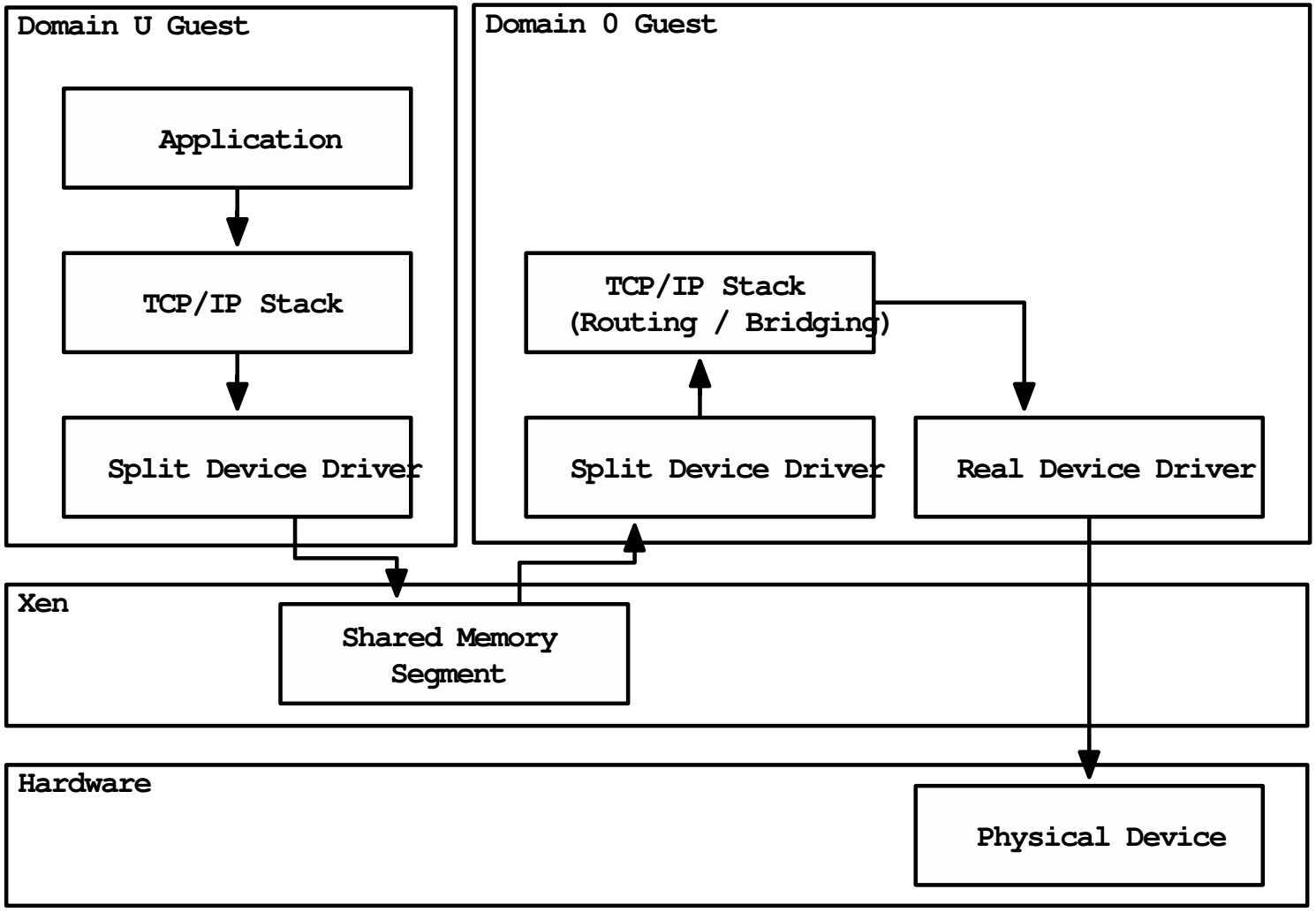
Use of privilege levels (rings) on paravirtualized X86 OS



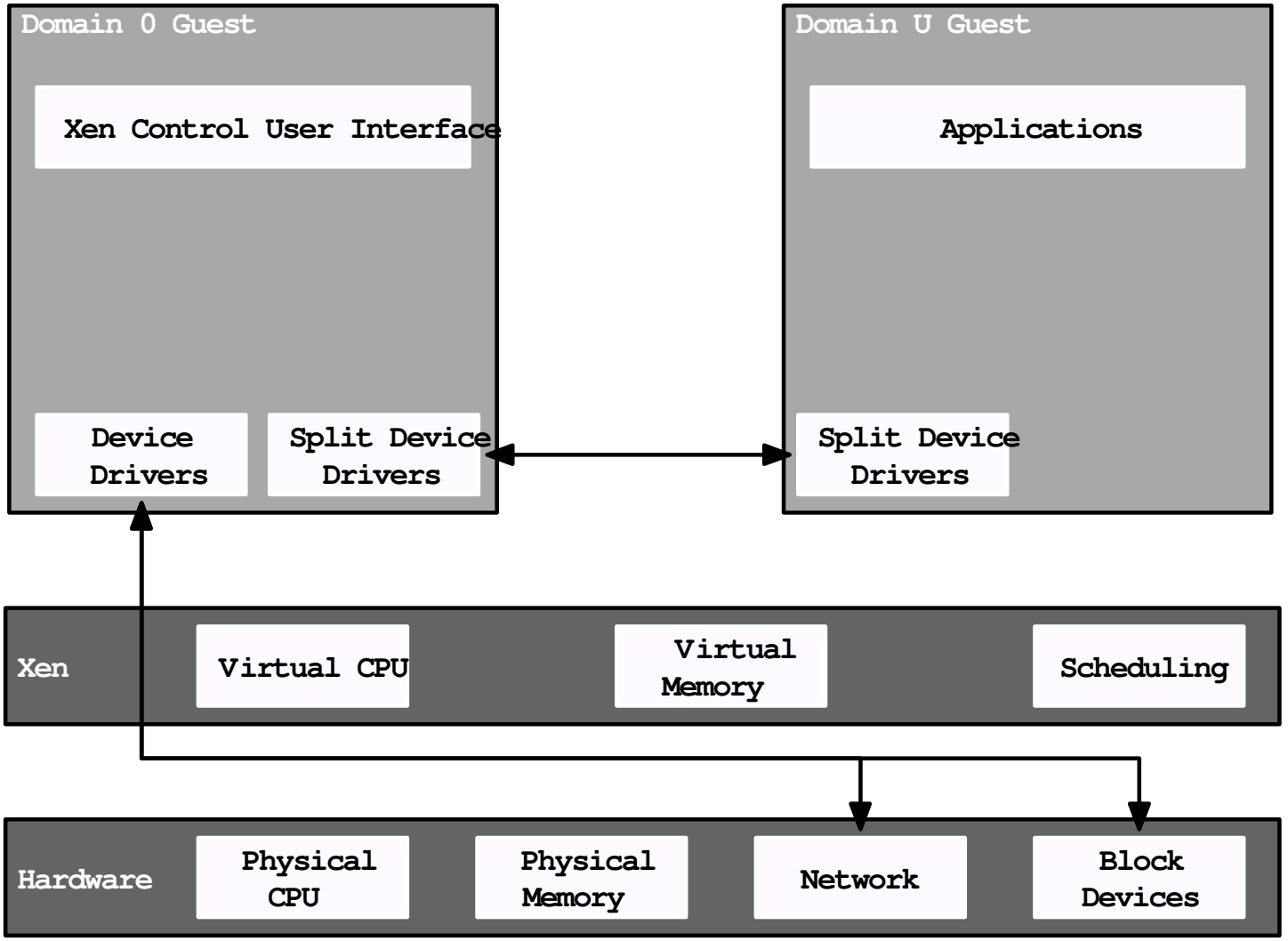
Use of privilege levels (rings) for AMD64/EMT64



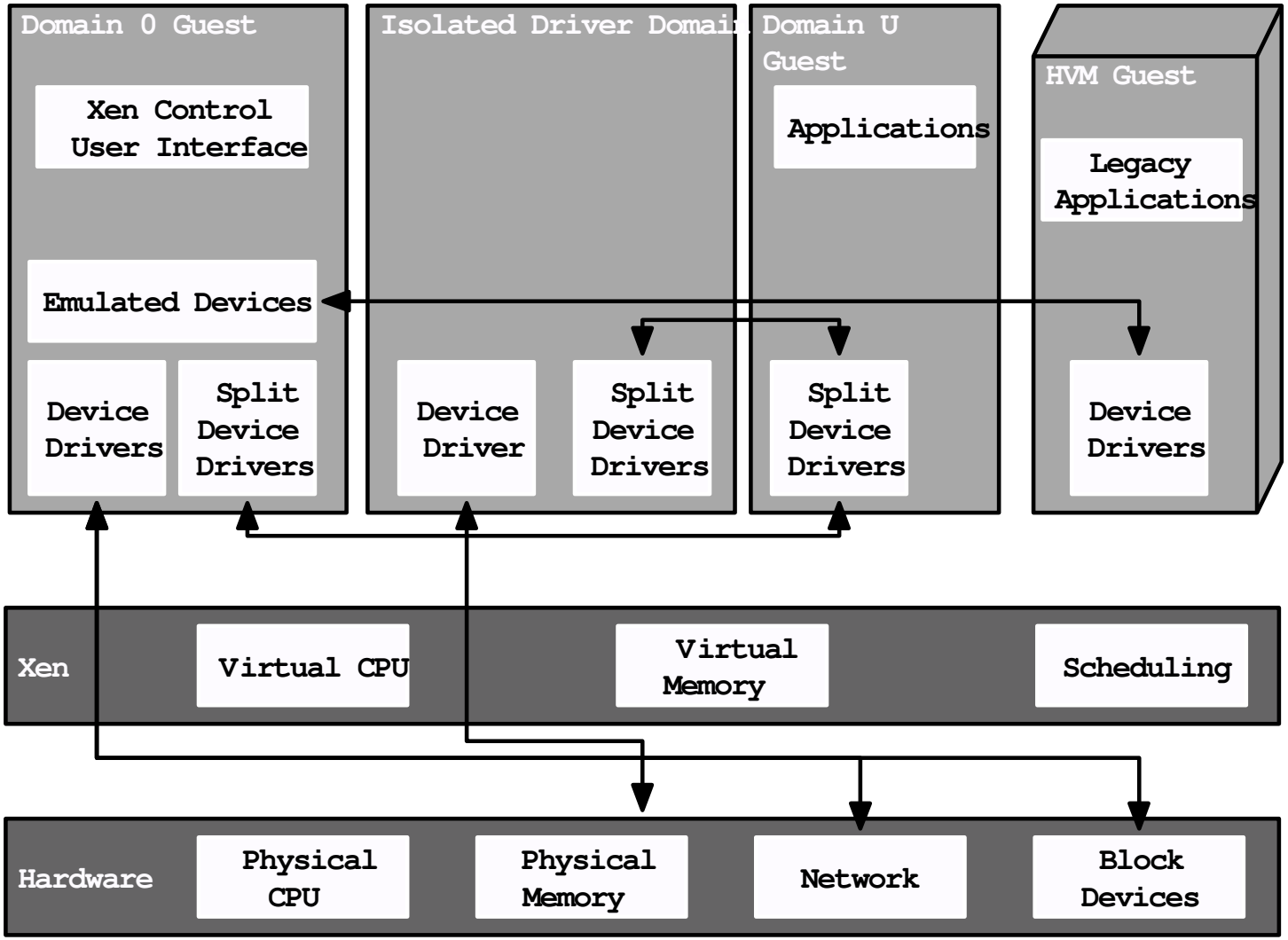
Packet Path from Unprivileged Domain



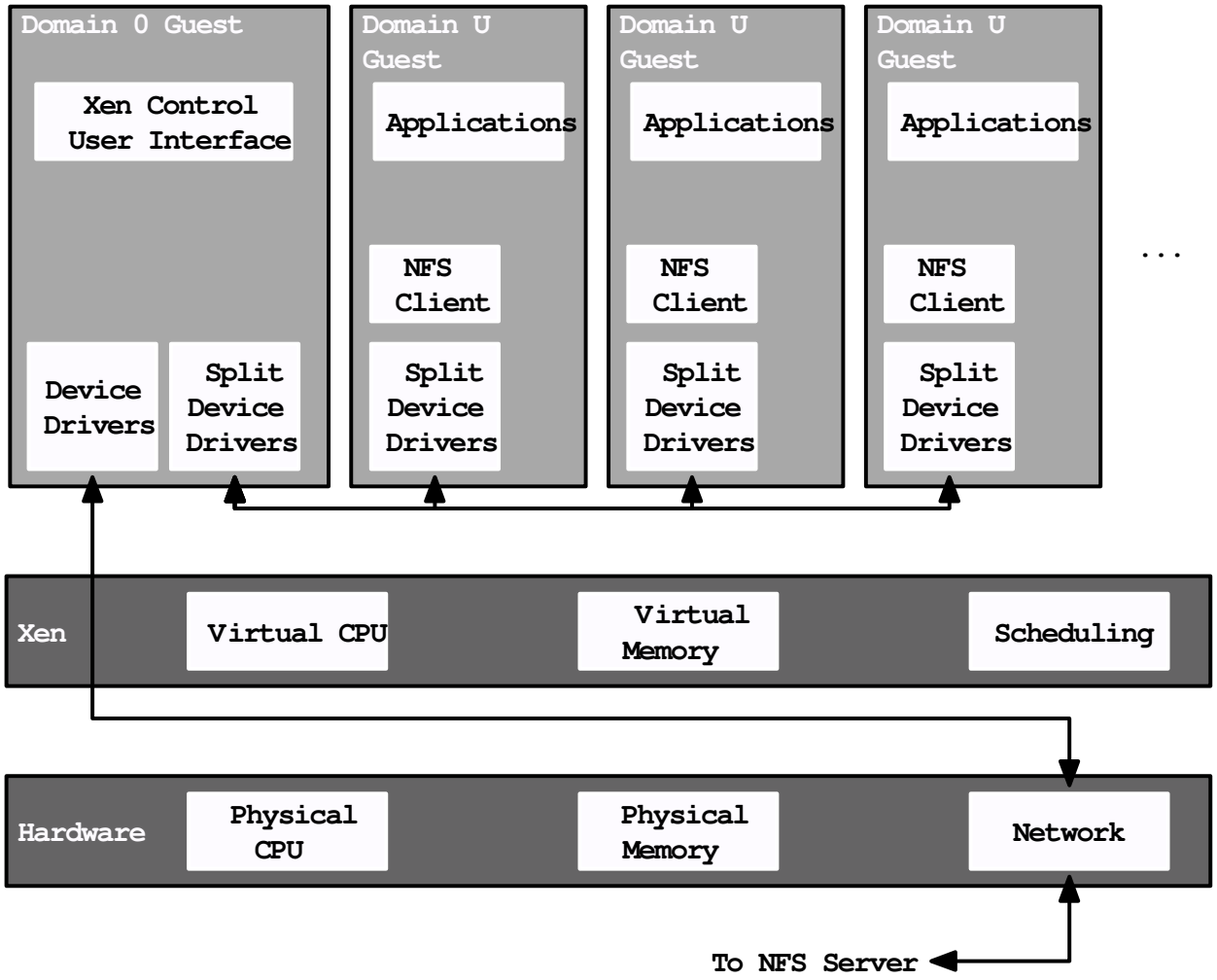
Peripheral architecture in Xen environment



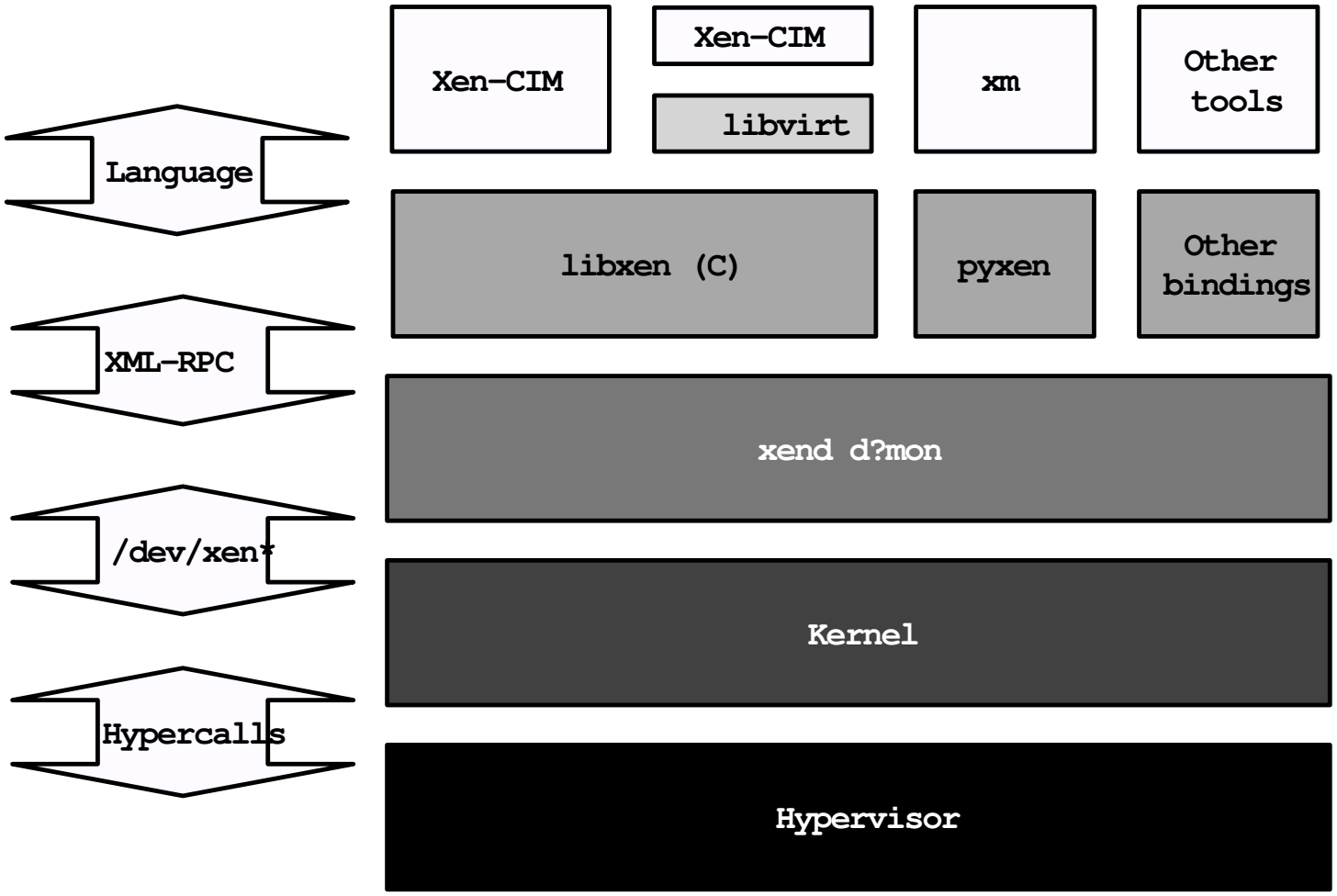
Xen – Unmodified Guest OS (HVM)



Xen – Fully paravirtualized/adapted Guest OS



Xen API Hierarchy



QEMU, GNU/Linux and more

- Good source of infomations about GNU/Linuxu porting, writing drivers and portabel applications are tutorial texts and presentation at **Free Electrons** server http://free-electrons.com/docs/
  - i.e. next resource specially for virtualization
    - Thomas Petazzoni / Michael Opdenacker: Virtualization in Linux