THE UNIVERSITY *of* EDINBURGH
informatics

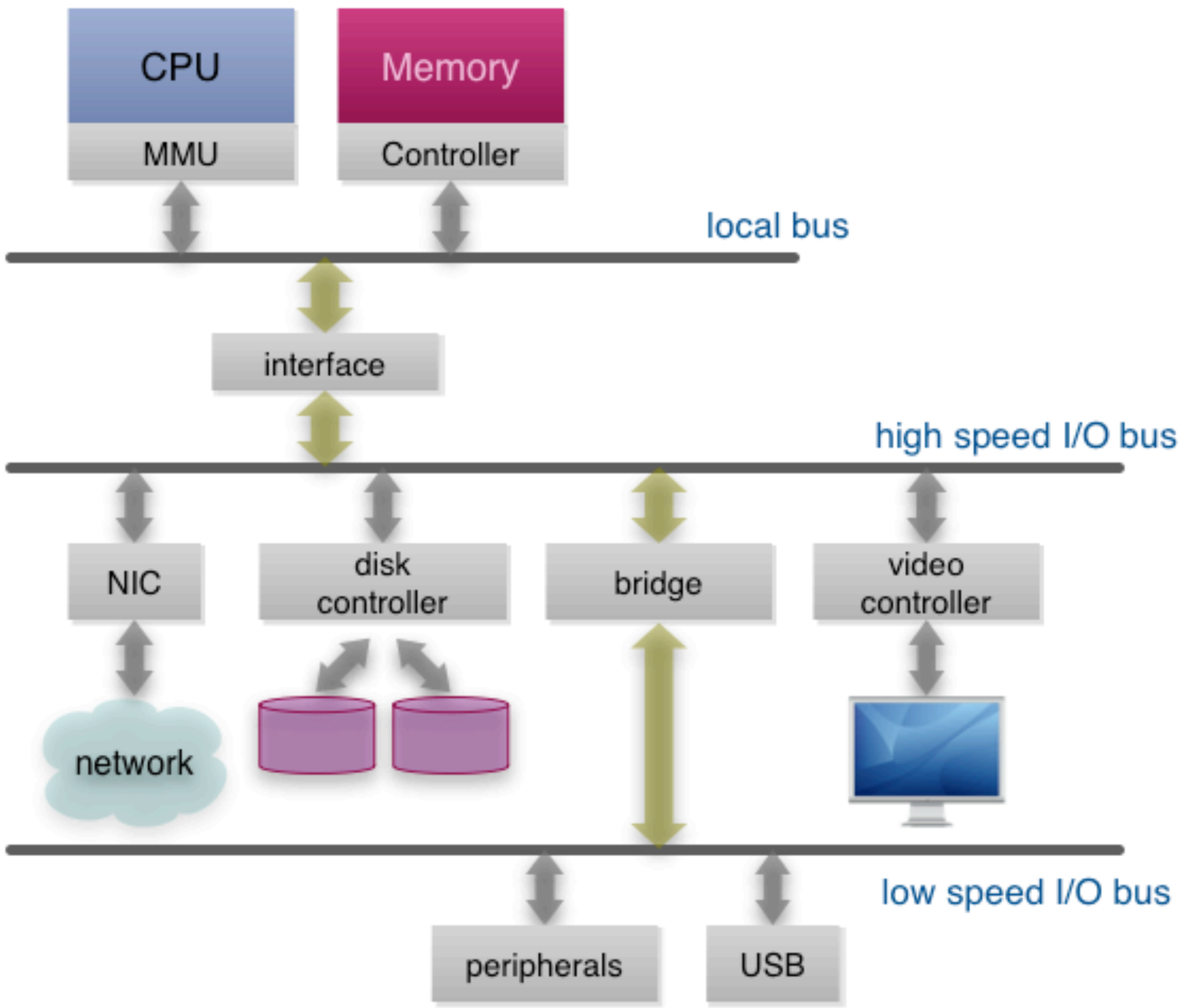
Extreme Computing

Behind the scenes: virtualisation

Overview

- One of the most important techniques for the separation of hardware, operating system, and applications
- Various instances of virtualisation used every day without even knowing (hey, it's virtual after all!)
- Started back in 1964 with IBM's CP-40, a "virtual machine/virtual memory time sharing operating system"
- Key ideas: abstraction and well-defined interfaces
  - These interfaces can be implemented differently for different platforms (think Java)
  - Or emulated by the host platform (think VMWare)
- We will focus on three types of virtualisation
  - CPU, memory, and device(I/O)

CPUs and computer architecture



What's in a CPU and how can we virtualise it?

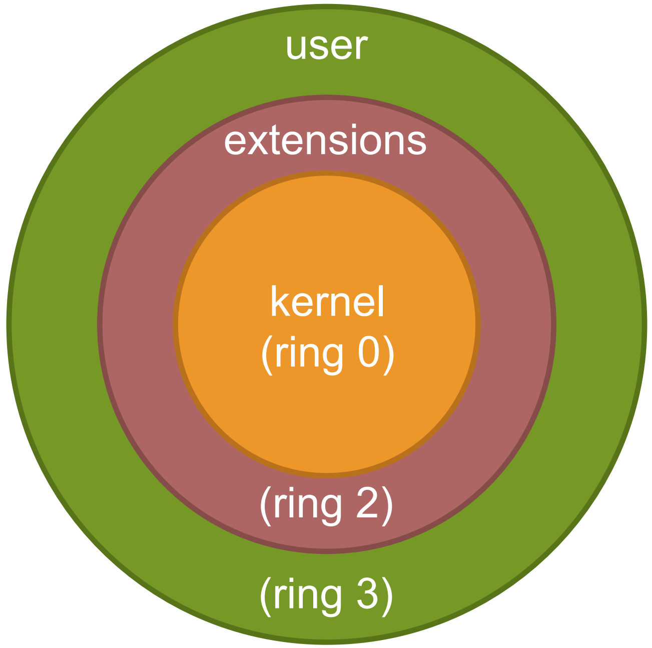
- It all comes down to one thing: the Instruction Set Architecture (ISA)
  - State visible to the programmer (registers, volatile memory)
  - Instructions that operate on the state
- Divided into two parts
  - User ISA used for developing/executing user programs (go wild, you can't break the system from here)
  - System ISA used mainly by the kernel for system resource management (careful here)
- Most CPU virtualisation techniques focus on the ISA
  - System ISA virtualisation, instruction interpretation, trap and emulate, binary translation, hybrid models

User ISA: state and instructions

- State captures the various components of the system
  - Virtual memory (physical, swap)
  - Special purpose registers (program counter, conditions, interrupts)
  - General purpose registers (this is the actual data that is manipulated)
  - ALU floating point registers (mathematical operations)
- Instructions capture the current parameters of each stage in the processor's pipeline
  - Typically: fetch, decode, access registers, memory, write-back
  - One instruction per stage
  - Mutiple instructions in the pipeline, at different stages

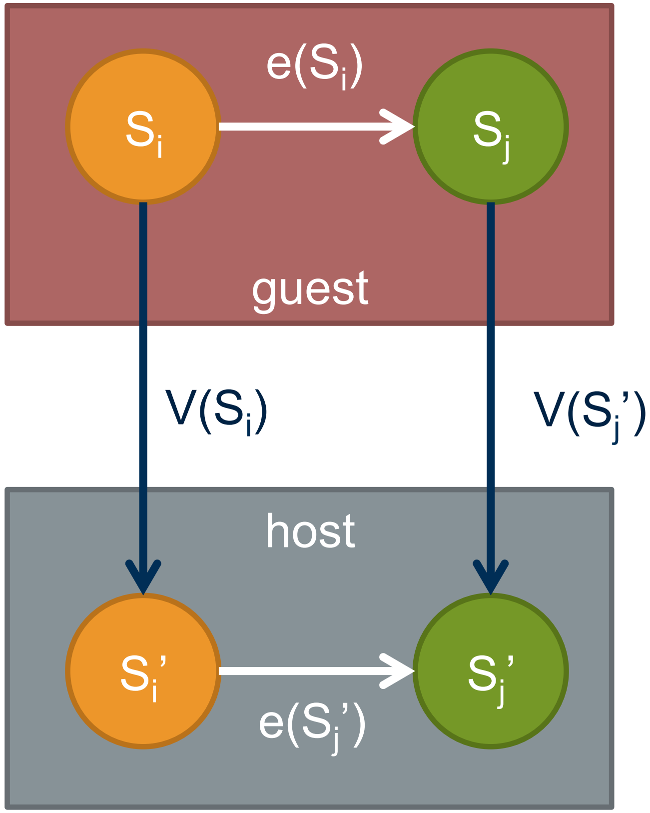
System ISA: where it all takes place

- Privilege levels (or rings)
- Control registers of the processor
- Processor and/or operating system traps and interrupts
  - Hard coded vectors (non-maskable interrupts and standard handlers)
  - Dispatch table (extension interrupt handlers)
- System clock
- Memory management unit
  - Page table, translation lookaside buffer
- Device I/O



kernel + extensions = system

The CPU virtualisation isomprphism

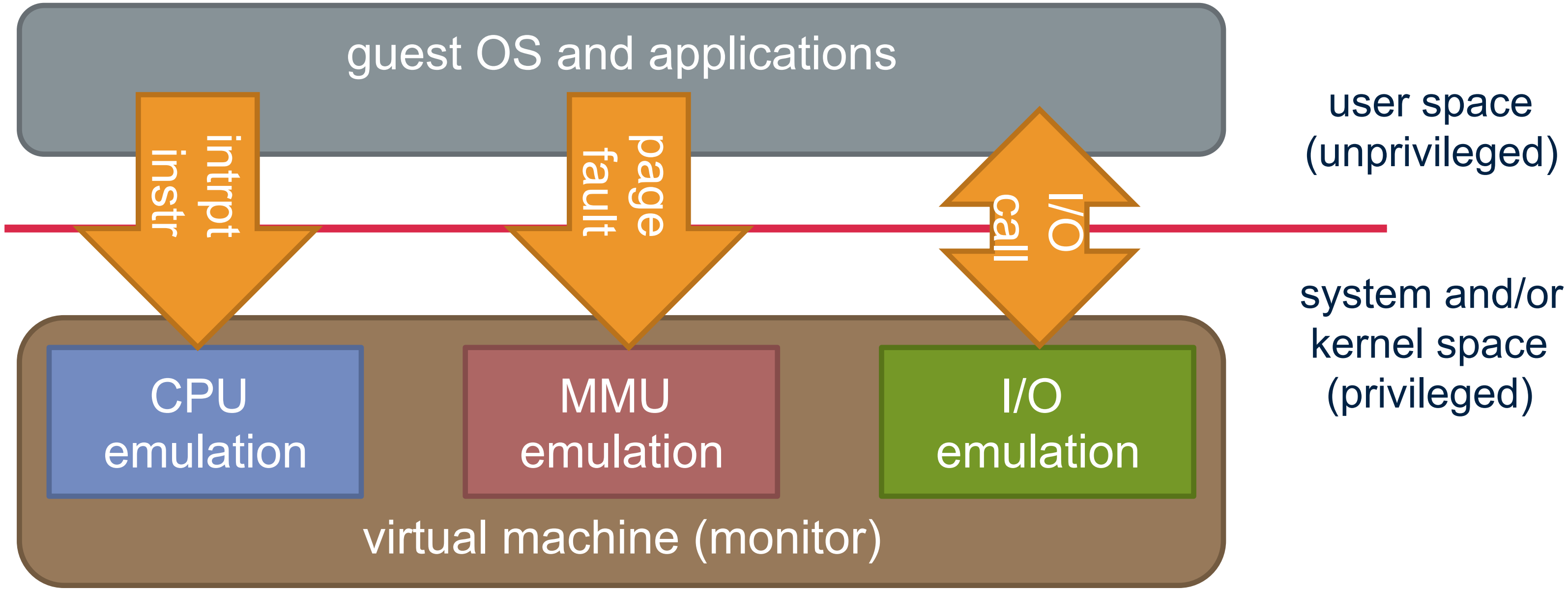


- Virtualisation is the construction of an isomorphism from guest state to host state
  - Guest state S_i is mapped onto host state *Si'* through some function $V()$: $V(S_i) = S_i'$
  - For every transformation $e()$ between states S_i and S_j in the guest, there is a corresponding transformation $e'()$ in the host such that $e'(S_i') = Sj'$ and $V(S_j) = S_j'$
  - Virtualisation implements $V()$ and the translation of $e()$ to $e'()$

Virtualising the System ISA

- Key concept: the virtualisation monitor (or hypervisor)
  - This is the actual implementation of the virtual machine
  - The guest assumes complete control of the hardware
  - But that is not possible—in fact, it's a security breach
  - So the monitor supervises the guest and virtualises calls to the guest's System ISA
  - Retargets them for the host
- Methodology is straightforward
  - Whenever the guest accesses the System ISA, the monitor takes over
  - Monitor maintains guest system state and transforms it whenever necessary
  - Guest system instructions are implemented as monitor functions affecting the host
  - Two-fold goal
    - Normal instructions are executed natively
    - Privileged instructions are isolated from the host

Trap and emulate



- Not all architectures support "trap and emulate" virtualisation
  - Most current CPUs have direct virtualisation hooks
- Trapping costs might be high (more calls than necessary)
- Virtual monitor runs at a higher privilege level
  - For instance, the Linux kernel only supports rings 0 (kernel) and 3 (user) though extensions like `kvm` solve the problem

Other types of CPU virtualisation

- Binary translation
  - Either compile programs to an intermediate representation and interpret them
    - Java (bytecode), llvm (virtual processor)
    - Implement the entire runtime multiple times for different platforms
  - Or, transform on-the-fly the natively compiled binary code
    - Very error-prone and hard to get right, especially when shifting between architectures
- Hybrid models
  - Solid parts of the system are binary translated (e.g., kernel functionality)
  - User code is trapped and emulated

But where is the monitor?

- The virtual machine monitor is yet another process
  - Shares the same virtual address space with the address space it is virtualising (!)
- As with CPU virtualisation, it handles specific interrupts (page faults)
- If using trap-and-emulate CPU virtualisation the situation is somewhat easier
  - The monitor only needs to be protected from guest accesses
  - Easy; run in host kernel/extension level
  - Monitor specific ranges of the virtual address space to identify if a memory request needs to be resolved or not; offload others to host OS
- For binary translation need a memory model distinguishing between host (priviliged, non-translated) and guest (unprivileged, translated) accesses
- Hardware-support: segmentation on x86 architectures
  - Monitor in dedicated memory region
  - Guest cannot see monitor's region

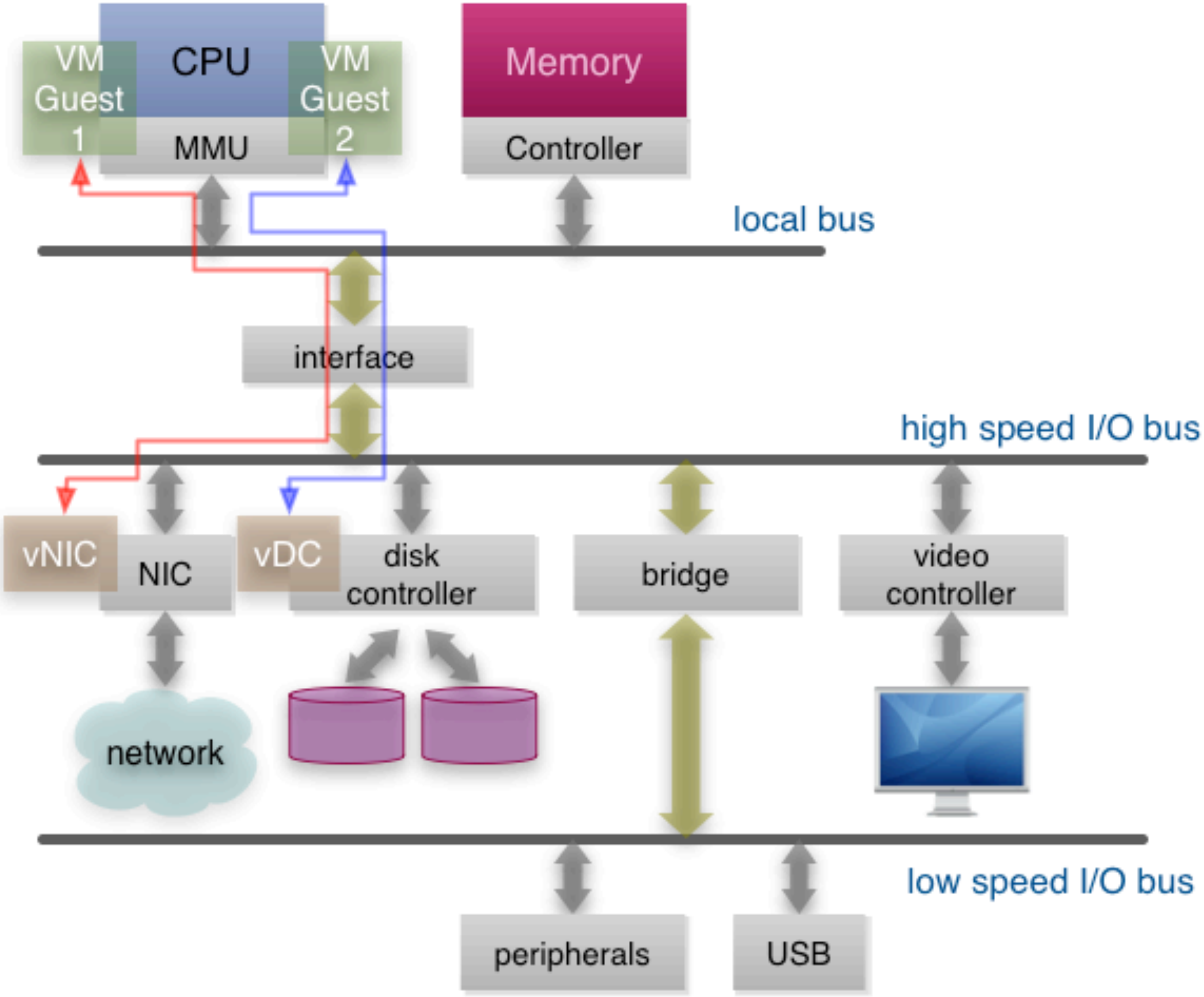
One step further out

- CPU virtualisation
  - Execute instructions developed for one CPU on another one
- Memory virtualisation
  - Allow multiple guest operating systems and their applications to see the same memory address space
  - Executed by a host operating system on a host CPU
- Both of them are a good start; but full-fledged systems access devices as well
  - A device is anything that can perform I/O
  - Hard-disk drives, displays, peripherals, you name it

Why virtualise I/O and how?

- Uniformity and isolation
  - A disk should behave like a single local disk regardless of whether it is remote or a RAID
  - Devices isolated from one another; they operate as if they were the only device around
- Performance and multiplexing
  - Let lower-level entities optimise the I/O path; they know how to do things better than explicit read/writes
  - Parallelise the process (e.g., when replicating data)
- System evolution and reconfiguration
  - Taking the system offline to connect a new drive, or repair a damaged one is no longer an option
- Techniques: direct access, emulation, paravirtualisation

Direct access



Virtualisation through direct access

- Advantages
  - No changes to guest, same operation is what it was designed for
  - Easy to deploy
  - Simple monitor: only implement drivers for the virtual hardware

- Disadvantages
  - Cannot happen without specialised hardware
  - Need to make the hardware interface visible to the guest
    - We just lost extensibility
  - Different hardware, different drivers
- Guest needs to cater for all possible drivers (not only the real ones, but the virtual ones as well!)
- Too much reliance on the hardware for software-related operations (e.g., scheduling, multiplexing, etc.)

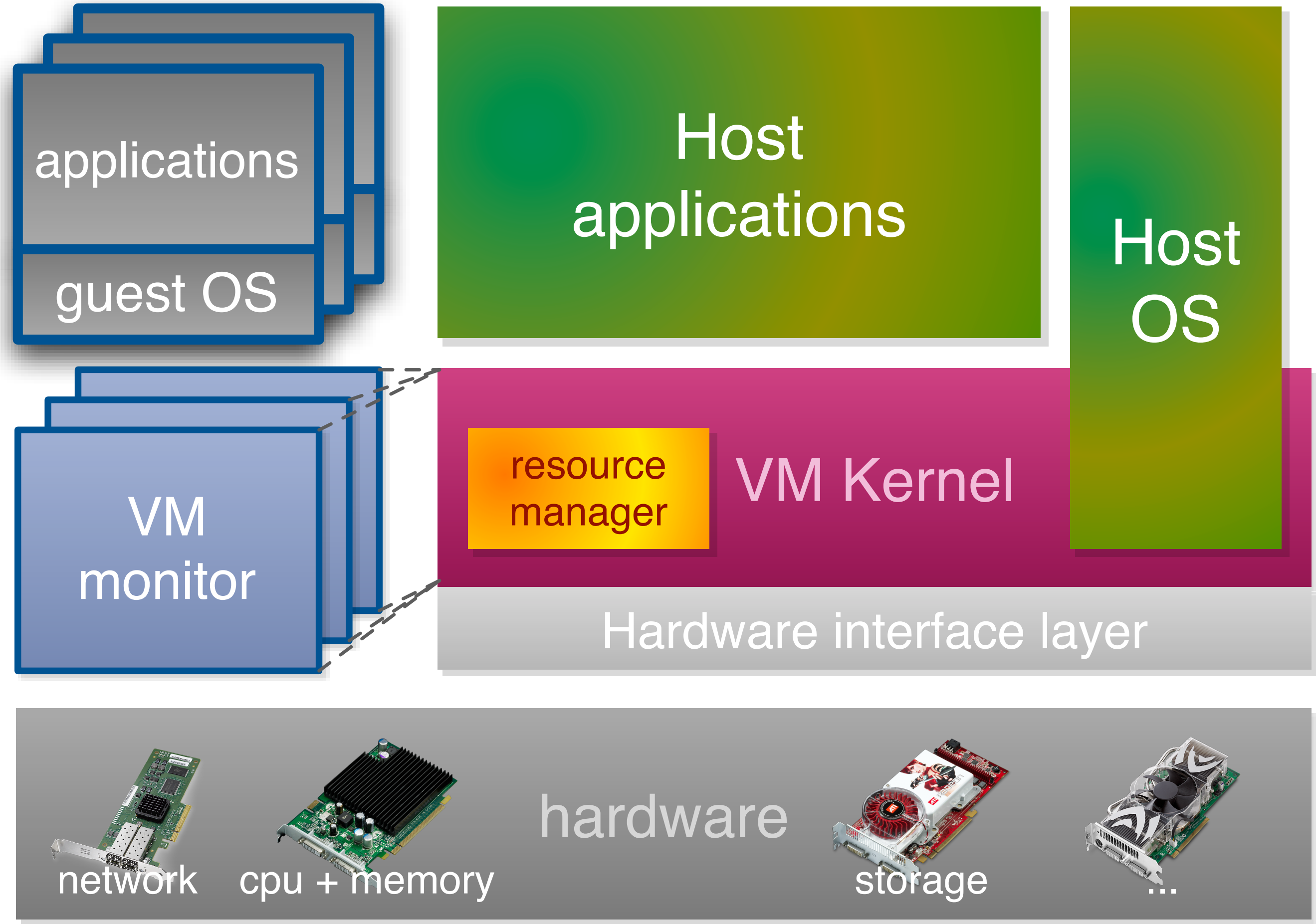
Device emulation

- Just as before, introduce an abstraction layer
  - Per class of device, e.g., for all disk drives
  - Implement the abstraction for different instances of the device e.g., drivers for disk interfaces, types of disk (HDD, solid state, …)
- Advantages
  - Device isolation
  - Stability: guest needs to operate just as before
  - Devices can be moved freely and/or reconfigured
  - No special hardware; all at the monitor level
- Disadvantages
  - The drivers need to be in the monitor or the host
  - Potentially slow: path from guest to device is longer
  - Possibility of duplicate effort: different drivers for the guest, different drivers for host

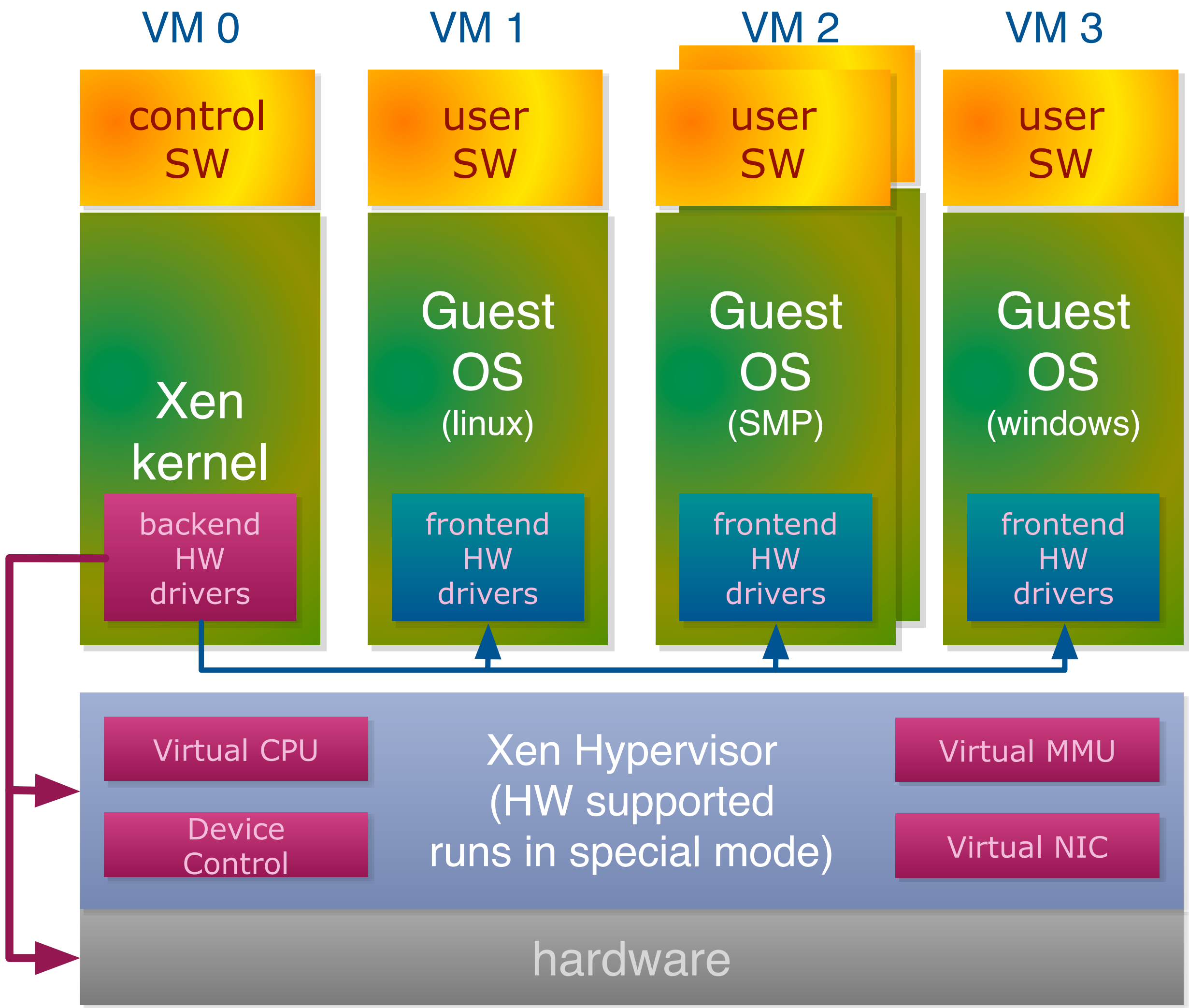
Paravirtualisation

- The solution most contemporary virtual machine monitors use
- Effectively, reverse the direction of the communication
  - Instead of trapping guest calls and emulating them by translating them for the host
    - Expose the monitor and allow guest to make monitor calls
    - Implement guest-specific drivers
    - Implement the drivers once for each device at the monitor
- Advantages
  - Monitor now becomes simpler (and simple usually equals fast)
  - No duplication
- Disadvantages
  - We still need drivers, but now drivers for the guest
  - Bootstrapping becomes an issue: can't host a guest operating system until there are drivers available

The design of VMWare ESX 2.04



The hybrid design of the Xen hypervisor



- Pararvirtualisation for Linux guests
- Hardware-virtualisation for Windows
- Single implementation of device drivers, single access to hardware

Summary

- Introduced virtualisation
- Discussed why it is necessary to use virtualisation
  - Abstraction of hardware from software
  - Ability to emulate other environments from specific CPUs and operating systems
- Presented different ways to virtualise various aspect of a computing system
  - Kernel, memory, devices
- Showed architectures used in contemporary virtualisation offerings