KIV Operační systémy

Virtualizace

Verze 1.0

Motivace

- Mějme konkrétní operační systém a konkrétní procesor
  - Programy zkompilované pro daný procesor a režim, ve kterém tento procesor běží, běží za takových podmínek nativně
  - Ale co když potřebujeme spustit program, který byl napsaný pro jiný režim procesoru, nebo jiný procesor, nebo dokonce pro jinou procesorovou architekturu?
  - Pak potřebujeme buď emulaci nebo virtualizaci

Emulace

- Softwarovým řešením vytváříme iluzi skutečného hardware
- Můžeme pak např. na ARMu spustit DOSBox – tj. staré programy pro x86
  - Nebo když potřebujeme spustit něco, co běží na hardware, který (už) nemáme k dispozici
- Jedná se sice o univerzální, ale výpočetně náročné řešení
  - V emulovaném prostředí lze také provádět virtualizaci

Virtualizace

- Virtualizace hw neemuluje, ale využívá hw, na kterém sama běží
  - Je proto výkonější než emulace
  - Ale také je limitovaná na programy, které byly zkompilované pro daný hw

Hypervizor

- Též známý jako Virtual Machine Monitor (VMM)
- (Hostitel) Vytváří a spouští virtuální stroje (hosty)
- Typ 1 – běží přímo na hw
  - Xen, VMWare ESX, Hyper-V
- Typ 2 – sám je hostován v OS
  - Virtual Box, WMVare Player

Mainframe

- Před érou PC dominovaly mainframy, které měly (na tehdejší dobu 50tých až 70tých let) velký výpočetní výkon, redundantní hw, I/O pro datově náročné aplikace..
- Jenomže nebyly vzájemně kompatibilní – jak tedy spustit software pro jeden mainframe na jiném?
  - IBM System/360 oddělila (virtualizovatelnou) architekturu od implementace
  - A tak bylo možné na jednom mainfraimu provozovat několik OS zároveň… až dnešních IBM zSeries, které se stále používají

PC

- PC je sice mnohem méně výkonné a spolihlivé než mainframe, ale zato je také daleko levnější
- Poměr cena/výkon dostala PC na „every desktop“
- Z PCs se dá navíc postavit cluster, tj. distribuovaný systém
  - V porovnání s mainframem stále levně
  - Software umí zajistit spolehlivost systému
  - A pomocí virtualizace pak lze efektivně vytvářet a spouštět virtuální stroje v takovém clusteru (dnes se říká cloudu)

V86 – motivace (1)

- Aneb jako to začalo na x86… viz první přednáška o MS-DOSu
- Začal se využívat protected-mode kvůli většímu adresnímu rozsahu a izolaci procesů, jenomže…
  - Některé programy nebyly přepsány do protected-mode, ale bylo potřeba je i nadále spouštět
  - Také bylo nutné ovládat zařízení,počítač nemá jenom jeden BIOS
    - Deska má svůj, grafická karta také, i síťová, SCSI, atd…
    - Protože BIOS inicializuje zařízení a x86 startuje v reálném režimu, BIOS obsahoval programy zkompilované jen pro reálný režim

V86 – motivace (2)

- BIOS neobsahuje jenom rutiny pro inicializaci hw, ale i rutiny pro jeho ovládání
  - Např. přepnutí video stránky SVGA podle VESA – viz první přednáška
  - A právě toho hojně využívaly ovladače – bylo pohodlnější zavolat již implementovanou funkci, než si ji napsat
- Protože se ale protected-mode od real-mode zásadně liší adresováním, nelze spustit v protected-mode program pro real-mode
  - =>x86 se musí přepnout do virtuálního režimu V86

V86 – popis

- Vznikl s 80386 po zkušenostech s implementací protected-mode u 80286
- Je to hw virtualizace 8086
  - Používá segmentaci jako real-mode, tj. 20-bitové adresy, ale ty už podléhají mechanismu stránkování v protected-mode
  - Pentium ještě přidalo pár vylepšení, Virtual 8086 Mode Enhancements – redukce režie s obsluhou přerušení
  - Na 64-bitových procesorech je dostupný už jenom v legacy-mode
  - Long-mode dokáže spustit program pro 8086 pomocí VT-x

V86 – princip

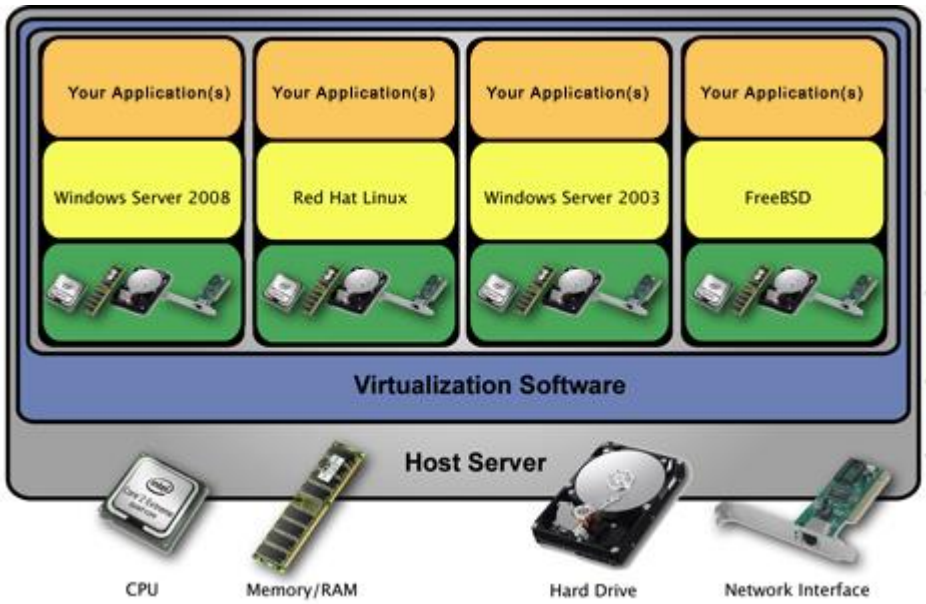
- Pokud bychom přepnuli procesor z protected-mode do real-mode, přijdeme o paměť a celý běžící OS by spadnul
- Proto se
  - Vytvoří 1MB velký paměťový prostor pro real-mode program, který poběží s CPL=3
  - Vytvoří se monitor, protected-mode task, s CPL=0
  - V okamžiku, kdy se real-mode program pokusí o privilegovanou operaci, řízení dostane monitor – CPU generuje vyjímku, ISR patří jádru
  - V okamžiku, kdy real-mode program volá službu OS, jádro převezme řízení – real-mode program generuje instrukci int, ISR patří jádru
  - Tj. v každém případě má jádro OS možnost vykonat privilegovanou operaci jak potřebuje, a real-mode program nic nepozná

V86 – důsledky

- Program běží rychle, protože jeho instrukce vykonává přímo CPU – tj. není emulován
- Není ale zcela pravda, že by real-mode program nemusel poznat, že je virtualizován
  - Některé instrukce to mohou prozradit, není-li použito VT-x nebo speciální technika, která takové instrukce z programu vyřadí před tím, než je spuštěn
  - Starý program to ale vědět nebude, protože v době jeho vzniku tyto instrukce neexistovaly

Problém dvou OS

- Jak zajistit, že jeden hostovaný OS nepoškodí jiný host. OS?
- Goldberg, „Formal Requirements for Virtualizable Third Generation Architectures“, 1974
  - Privilegované instrukce – CPU generuje vyjímku, jakmile se pokusí o vykonat instrukci, která neodpovídá CPL hostovaného OS
  - Sensitive instr. – mění hw konfiguraci a jejich výsledek závisí na aktuální hw konfiguraci



http://www.datahive.ca/data_centre_virtualization.html

Privilegované vs. sensitive

- Efektivně a bezpečně lze virtualizovat pouze tehdy, jsou-li sensitive instrukce podmnožinou privilegovaných instrukcí
  - Do příchodu Intel VT-x a AMD-V toto nebylo na x86 splněno
    - Např. SMSW byla sensitivní, ale ne privilegovaná
    - Ideově viz IBM VM/370 OS pro mainframe
  - Non-sensitive instrukce jsou vykonávány přímo CPU – jejich virtualizace má zanedbatelnou režii
  - Sensitive-instrukce – pokus o jejich vykonání generuje vyjímku, která se musí obsloužit – tj. zde dochází k emulaci v rámci virtualizace, a to je pomalé

Paravirtualizace

- Nemáme-li k dispozici obdobu VT-x, jedním z možných řešení je modifikovat hostovaný OS tak, aby nepoužíval instrukce které jsou sensitive, ale ne privilegované
  - Hostovaný OS si je vědom, že mezi ním a hw běží ještě tzv. hypervizor
  - Dostaneme výkonnostní potenciál virtualizace, ale…
  - Co se stane, když se nám do OS dostane a spustí program, který bude tyto zakázané instrukce obsahovat?
    - Bezpečnostní problém?

Binární překlad

- Aneb na čem byl založený business-plan VMware, který Intel VT-X a AMD-V zničily
- Než je hostovaný program spuštěný, je analyzovaný a všechny sensitivní, ale neprivilegované instrukce se nahradí sekvencemi instrukcí, které dělají to samé, ale bez nežádoucích vedlejších efektů
  - Dále je možné nahradit i ty sekvence instrukcí, které jinak vedou k emulaci – tj. když privilegovaná instrukce generuje vyjímku
  - Je to netriviální záležitost, protože nahrazovaná a nahrazující sekvence instrukcí nemusí mít stejnou velikost a v nahrazované sekvenci může být i cíl skoku

Binární překlad

- Aneb na čem byl založený business-plan VMware, který Intel VT-X a AMD-V zničily
- Než je hostovaný program spuštěný, je analyzovaný a všechny sensitivní, ale neprivilegované instrukce se nahradí sekvencemi instrukcí, které dělají to samé, ale bez nežádoucích vedlejších efektů
  - Dále je možné nahradit i ty sekvence instrukcí, které jinak vedou k emulaci – tj. když privilegovaná instrukce generuje vyjímku
  - Je to netriviální záležitost, protože nahrazovaná a nahrazující sekvence instrukcí nemusí mít stejnou velikost a v nahrazované sekvenci může být i cíl skoku

Binární překlad – int 10h

- Mějme program pro real-mode, který se snaží změnit mód obrazovky do textového režimu CGA 80x25x16/8

  mov ax, 3

  int 10h

- Tento kód vyžaduje emulaci grafické karty, takže bychom ho mohli rovnou nahradit sekvencí, která volá rovnou náš emulátor gr. karty, aniž bychom museli nejdříve přepínat kontext

  mov ax, 3

  pushf

  call EmulatedISR10h

Binární překlad – skok

- Mějme následující smyčku s privilegovanou instrukcí HLT, která nechá jádro CPU zastavené, dokud není signalizováno jinak

```
db flag
  ….
@skok:
    HLT                                   ;opcode HLT je 0xF4 – tj. 1B
    test byte ptr [flag], 0
    jz @skok:
```

- HLT vyžaduje CPL=0, jenomže, když se jí pokusíme nahradit pomocí call, máme problém – instrukce call má sice opcode také veliký 1B, jenomže následuje alespoň jeden další byte cíle skoku
  - Nahrazení HLT s call vyžaduje úpravu parametrů instrukcí, které adresují paměť – jmp, jnz, call, test, mov, inc, dec…. Toto není triviální

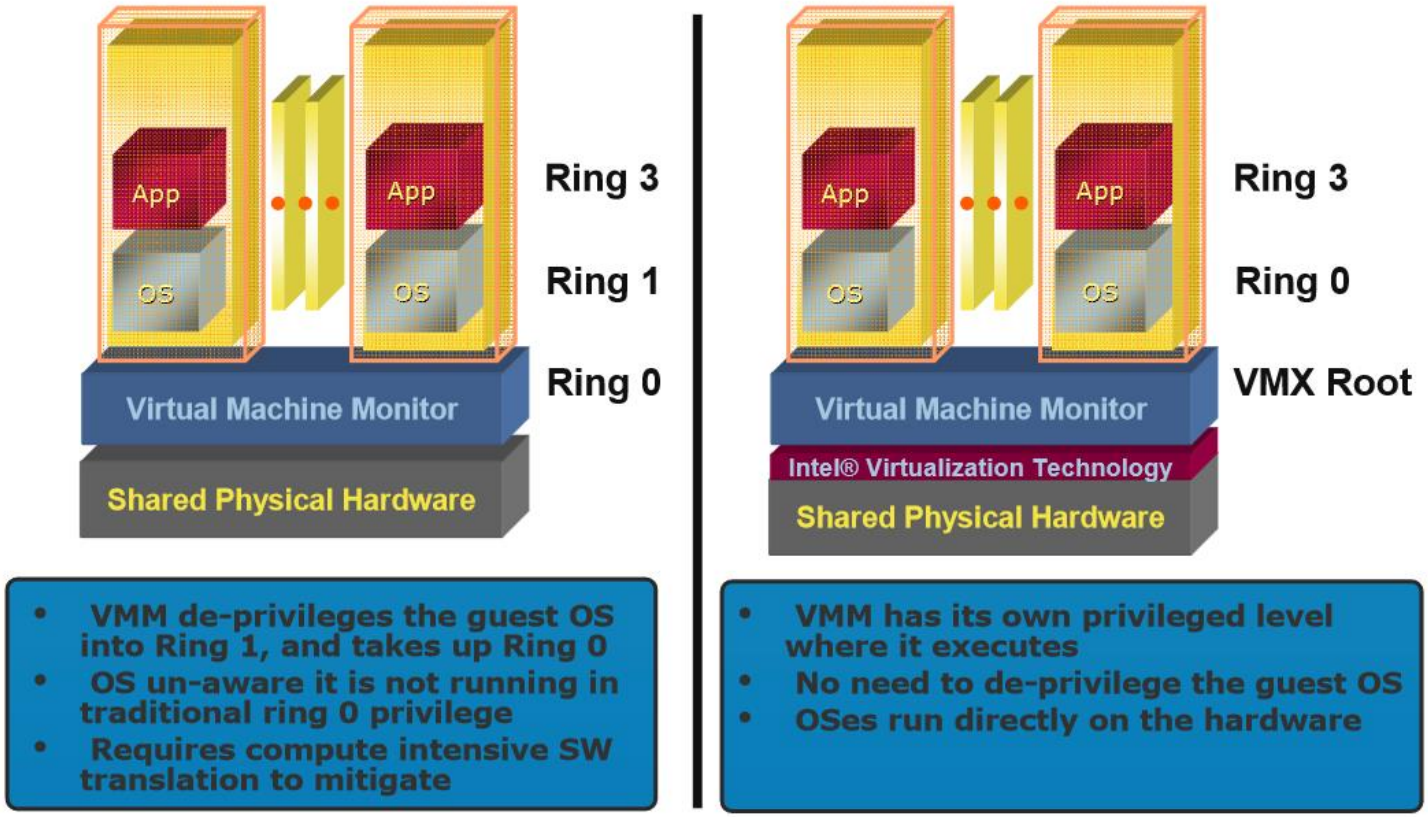
Binární překlad – pasti

- Mějme program, který obsahuje ochranu proti zpětnému inženýrství, pro jednoduchost uvažujme následující 3 byty
  - db 0ebh, 0ffh, 0c0h
  - Jedná se o instrukce jmp -1 a inc eax, které sdílejí byte 0ffh
    - 0ffh je totiž -1 jako relativní adresa skoku a zároveň je to opcode instrukce inc
    - Trik je v tom, že se disassembler po jmp -1 nevrátí o 1 byte zpět, a proto pro něj bude mít další instrukce opcode 0c0h a diassemblovaný kód pak po těchto třech bytech nebude dávat smysl
    - Anebo by se disassembler mohl vrátit o 1B zpět, ale to už vyžaduje heuristiku simulující chování procesoru

VT-x

- Cílem je eliminovat potřebu paravirtualizace a binárního překladu
- Procesor běží ve dvou režimech, takže odpadá potřeba měnit CPL – nicméně je třeba minimalizovat přechody mezi nimi
  - VMX root; přechod VM Entry
  - VMX non-rooot – hostovaný OS; přechod VM Exit
- Hostovaný OS bez jakékoliv úpravy běží ve VMX non-root režimu. a ani z žádného stavového bitu to nepozná. Jakmile se pokusí o operaci, kterou nemá dovolenu provést, dojde k tzv. VMX-transition. Řízení přebere hypervizor, který provede, co je třeba, ve VMX-root režimu procesoru.

Pre & Post Intel VT-x



- VMM de-privileges the guest OS into Ring 1, and takes up Ring 0
- OS un-aware it is not running in traditional ring 0 privilege
- Requires compute intensive SW translation to mitigate

- VMM has its own privileged level where it executes
- No need to de-privilege the guest OS
- OSes run directly on the hardware

https://software.intel.com/sites/default/files/m/0/2/1/b/b/1024-Virtualization.pdf

TLB

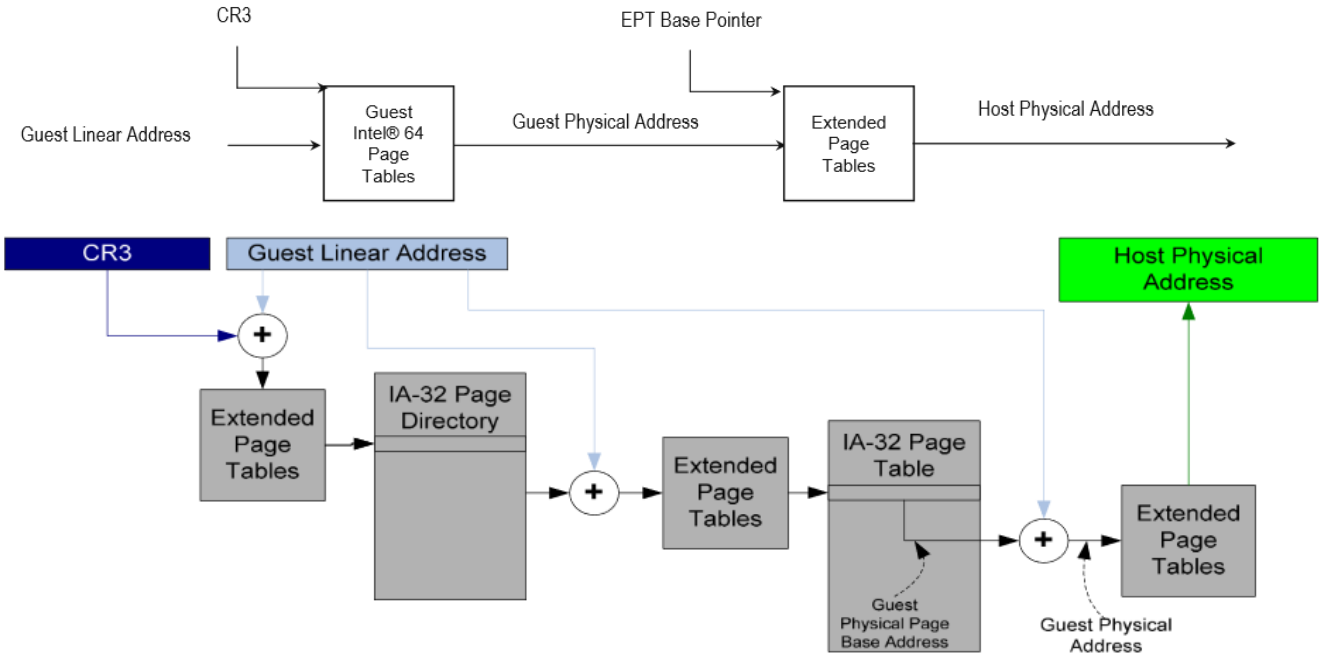
- Pokud by se při každém VMX transition měla v rámci bezpečnosti vyprázdnit TLB, mělo by to vliv na výkonost
  - Translation look-aside buffer – viz stránkování, druhá přednáška
  - A první generace VT-x to i dělala
  - V rámci vylepšení má každý VMX non-root host VPID – Virtual Processor ID
  - Položky v TLB mají VPID, takž se ví, komu patří a není nutné TLB vyprázdnit

Stránkování

- Shadow page-table jsou tabulky stránek hostujícího OS
- Tabulka stránek hostujícího OS je bez VT-x read-only, což umožní zachytit pokus o její modifikaci a následně synchronizovat shadow verzi
  - Jenomže pokus o zápis by generoval vyjímku, a to je pomalé
- VT-x má proto koncept zanořených/rozšířených tabulek stránek, který toto eliminuje

Stránkování

- Host používá tabulku stránek, jak byl zvyklý, ale fyzická adresa hosta se ještě přes rozšířenou tabulku, tj. zanoření, převede na fyzickou adresu hostitele

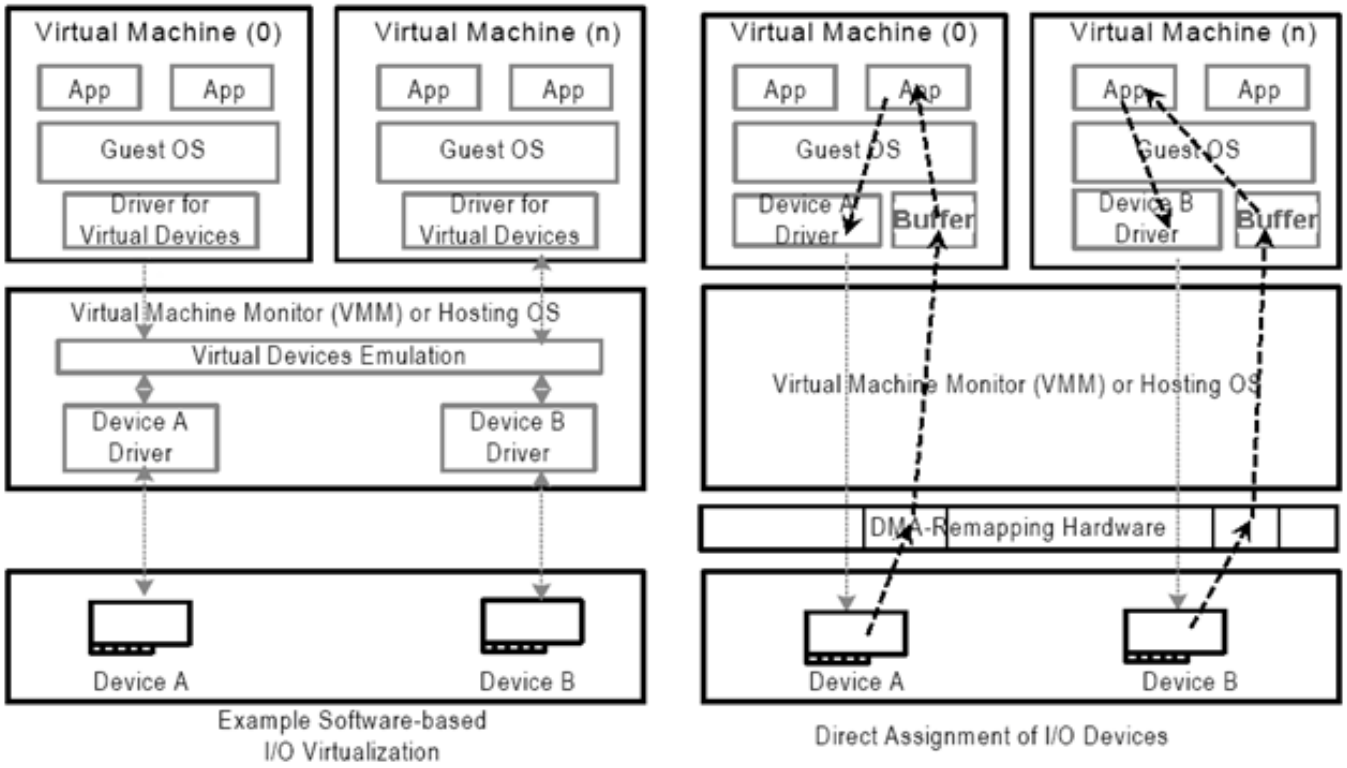


https://software.intel.com/sites/default/files/m/0/2/1/b/b/1024-Virtualization.pdf

VT-d/VT for Directed I/O

- K výkonnostním penaltám dochází ještě při přístupu k I/O zařízení – opět díky VMX-Transition do VMX-root
- VT-d umožňuje přemapování IRQ a definuje další verzi architektury DMA
  - I/O zařízení lze přiřadit přímo konkrétnímu hostu
  - Vyžaduje extended xAPIC

VT-d/VT for Directed I/O



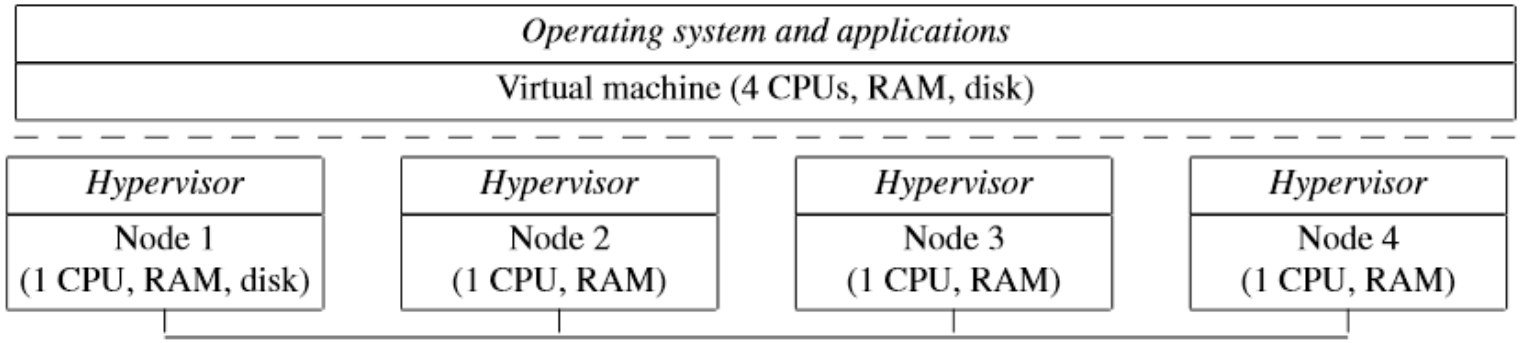
https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices

Virtualization for Aggregation

- Nesnažíme se virtualizovat jeden počítač pro několik hostů, ale naopak se snažíme virtualizovat několik počítačů pro jednoho hosta
- Levně můžeme postavit počítač s enormním množstvím RAM a procesorových jader z běžně dostupných komponent
- Hostovaný OS uvidí jenom (virtuální) SMP
- S vSMP padají náklady na údržbu clusterů, na portování a vývoj programů pro distribuované prostředí

VfA – jak?

- Na každém počítači VfA vSMP běží VMM, který používá např. VT-x a komunikuje s ostatními VMM
- Když zachytí přístup k něčemu, co se nachází na počítači s jiným VMM, zasíláním zpráv „přesměruje“ zachycený požadavek jinému VMM, který ho vyřídí na svém HW



https://www.usenix.org/legacy/event/usenix09/tech/full_papers/chapman/chapman.pdf

Rootkit

- Máme-li k dispozici tak perfektní virtualizaci, jak složité by s ní bylo vytvořit rootkit?

1. Inicializace VT-x
2. Vytvoření VM a VMM
3. Zkopírování hostitelského OS do VM
4. Předání řízení do VM
5. Ukončení činnosti v hostitelském OS, nyní již běžícím jako hostu
6. Při VMX Transition do VMX root se aktivuje rootkit a ví vše, co se děje v hostu

Detekce rootkitu

- Sice není k dispozici oficiálně dokumentovaný stavový bit, který by host mohl použít, ale…
- Např. CPUID vždy způsobí VM Exit
  - Tj. má podstatně větší latenci, když je VT-x aktivní!