

B4B350SY: Operační systémy

Virtualizace

Michal Sojka

`michal.sojka@cvut.cz`



21. prosinec, 2017

1 Úvod

2 Virtualizace celého systému

- Virtualizace CPU
- Hardwarově asistovaná virtualizace
- Virtualizace vstupu a výstupu

3 Kontejnery

Obsah

1 Úvod

2 Virtualizace celého systému

- Virtualizace CPU
- Hardwarově asistovaná virtualizace
- Virtualizace vstupu a výstupu

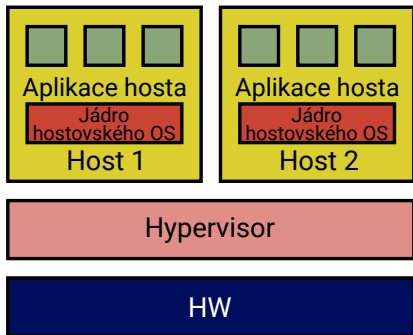
3 Kontejnery

Co je virtualizace?

- Základní myšlenka: abstrakce hardwaru počítače a jeho částečná emulace v SW
 - Operační systém, který běžel na fyzickém hardwaru běží na virtuálním HW (**virtuální stroj, VM**)
- Hlavní komponenty
 - **Hostitel** (host) – fyzický hardware na kterém vše běží
 - **Hypervizor a virtual machine monitor (VMM)** – SW implementující virtuální hardware
 - **Host** (guest) – SW (typicky OS) běžící na virtuálním HW
- Jeden hostitel typicky může hostit více hostů najednou
- Používaná od 70. let, zejména na mainframech firmy IBM
 - Popek a Goldberg definovali požadavky pro virtualizaci počítačové architektury v r. 1974
 - Architektura x86 je plně virtualizovatelná od r. 2005

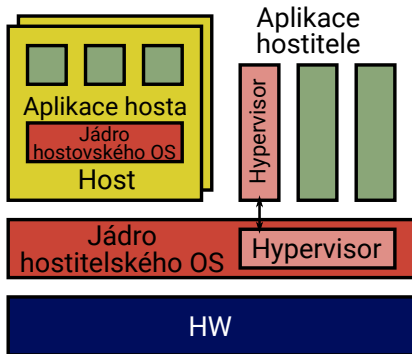
Typy hypervizorů

- Hranice mezi typy není vždy zřejmá. I mnohé hypervizory typu 1 mohou spouštět nativní aplikace – např. konfigurační nástroje.



Typ 1 – nativní

- Příklady: VMware ESX, Citrix XenServer



Typ 2 – hostovaný

- Příklady: Oracle VirtualBox, VMware Workstation, Parallels Desktop, Qemu

Výhody virtualizace

- **Izolace** virtuálních strojů (VM) mezi sebou
 - V ideálním případě se útok či virus nerozšíří z jednoho VM do ostatních
- **Nezávislost** softwaru na hardwaru konkrétního počítače
- Běh **více různých OS** na jednom počítači
- Možnost **pozastavení** běhu VM
 - Pozastavenou VM lze zkopírovat či přesunout na jiný počítač a **pokračovat** v běhu jinde.
- Možnost **živé migrace** – běžící VM je přesunut na jiného hostitele bez přerušení přístupu uživatelem
- Vytváření **šablon**
 - šablona = obraz OS s aplikací, distribuce všeho dohromady zákazníkům, možnost spustit víckrát
- Při **vývoji OS** „nepadá“ celý počítač
 - Lze použít pro vývoj částí OS nezávislých na HW
 - Pro vývoj ovladačů většinou nevhodné
 - Reverzní inženýrství...

Cloud computing

- Virtualizace je základem pro tzv. „cloud computing“
- Cloud většinou umožňuje uživatelům vzdálenou správu virtuálních strojů
 - API a nástroje pro komunikaci s infrastrukturou cloudu
 - Vytváření virtuálních strojů a jejich konfigurace

Je virtualizace skutečně potřeba?

- Hlavní výhody virtualizace z předchozí stránky jsou zároveň vlastnosti požadované od každého OS
 - OS izoluje aplikace (procesy) mezi sebou
 - Aplikace jsou díky OS nezávislé na HW
- Dnešní popularita virtualizace je důsledek nedokonalosti běžných OS
 - Kdyby byl OS dokonalý, nepotřebujeme pouštět víc OS
 - ...

Je virtualizace skutečně potřeba?

- Hlavní výhody virtualizace z předchozí stránky jsou zároveň vlastnosti požadované od každého OS
 - OS izoluje aplikace (procesy) mezi sebou
 - Aplikace jsou díky OS nezávislé na HW
- Dnešní popularita virtualizace je důsledek nedokonalosti běžných OS
 - Kdyby byl OS dokonalý, nepotřebujeme pouštět víc OS
 - ...

Základní pravidlo softwarového inženýrství

Každý problém softwarového inženýrství lze vyřešit přidáním vrstvy abstrakce (layer of indirection). Jedinou výjimkou je problém příliš mnoha vrstev abstrakce.

—David J. Wheeler

Proč jsou běžné OS nedokonalé a co s příliš mnoha vrstvami abstrakce?

- V minulosti byla preferována rychlost před bezpečností (izolací)
 - ⇒ monolitická jádra OS: stačí jediná zranitelnost a celý systém je kompromitován
- OS jsou složité
 - Změna architektury OS by byla komplikovaná a nefungovaly by staré aplikace
 - Bylo jednodušší emulovat (virtualizovat) HW než hledat řešení v (uzavřených) OS

Proč jsou běžné OS nedokonalé a co s příliš mnoha vrstvami abstrakce?

- V minulosti byla preferována rychlost před bezpečností (izolací)
 - ⇒ monolitická jádra OS: stačí jediná zranitelnost a celý systém je kompromitován
- OS jsou složité
 - Změna architektury OS by byla komplikovaná a nefungovaly by staré aplikace
 - Bylo jednodušší emulovat (virtualizovat) HW než hledat řešení v (uzavřených) OS

Řešení

- Jedné vrstvy abstrakce se zbavíme jejím přesunem do hardwaru: HW akcelerace virtualizace ⇒ lepší HW (viz dále)
- Lepší architektura OS – mikrojádru (začínají se prosazovat v mnoha aplikacích, zejména pokud jde o bezpečnost)

Typy virtualizace

- **Virtualizace celého systému** – hostovaný systém neví, že běží na virtualizovaném systému (viz dále)
- **Paravirtualizace** – virtualizovaný systém ví, že neběží na skutečném HW a „dobrovolně“ spolupracuje s hypervizorem (tj. explicitně volá jeho služby)
- **Emulace celého systému** – vše je emulováno v SW, včetně vykonávání instrukcí. Např. Qemu umí vykonávat programy pro ARM na x86.
- **Virtualizace běhového prostředí programu** – Java VM, C# VM (mimo rámec tohoto předmětu)
- **Kontejnerizace aplikací** – viz dále

Obsah

1 Úvod

2 Virtualizace celého systému

- Virtualizace CPU
- Hardwarově asistovaná virtualizace
- Virtualizace vstupu a výstupu

3 Kontejnery

Virtualizace CPU

- Klasické CPU vykonává kód ve dvou režimech:
 - **uživatelském** (uživatelské aplikace, x86: Ring 3)
 - **privilegovaném** (jádro OS, x86: Ring 0)
- Při virtualizaci:
 - nemůžeme nechat vykonávat hostované jádro v privilegovaném režimu – nebyla by zajištěna izolace VM mezi sebou
 - potřebujeme implementovat **virtuální uživatelský** a **virtuální privilegovaný** režim
 - **skutečný privilegovaný režim** použijeme pro hypervizor
 - uživatelský i privilegovaný virtuální režim běží ve **skutečném uživatelském režimu** procesoru

Virtualizace CPU

- Klasické CPU vykonává kód ve dvou režimech:
 - **uživatelském** (uživatelské aplikace, x86: Ring 3)
 - **privilegovaném** (jádro OS, x86: Ring 0)
- Při virtualizaci:
 - nemůžeme nechat vykonávat hostované jádro v privilegovaném režimu – nebyla by zajištěna izolace VM mezi sebou
 - potřebujeme implementovat **virtuální uživatelský** a **virtuální privilegovaný** režim
 - **skutečný privilegovaný režim** použijeme pro hypervizor
 - uživatelský i privilegovaný virtuální režim běží ve **skutečném uživatelském režimu** procesoru
- **Důsledek:** Virtuální stroj je z pohledu OS/hypervizoru velmi podobný běžnému procesu.

Virtualizace CPU

- Klasické CPU vykonává kód ve dvou režimech:
 - **uživatelském** (uživatelské aplikace, x86: Ring 3)
 - **privilegovaném** (jádro OS, x86: Ring 0)
- Při virtualizaci:
 - nemůžeme nechat vykonávat hostované jádro v privilegovaném režimu – nebyla by zajištěna izolace VM mezi sebou
 - potřebujeme implementovat **virtuální uživatelský** a **virtuální privilegovaný** režim
 - **skutečný privilegovaný režim** použijeme pro hypervizor
 - uživatelský i privilegovaný virtuální režim běží ve **skutečném uživatelském režimu** procesoru
- **Důsledek:** Virtuální stroj je z pohledu OS/hypervizoru velmi podobný běžnému procesu.
- *Jak zajistit, že při systémových voláních z virtuálního uživatelského režimu přejdeme do virtuálního privilegovaného režimu a ne do skutečného privilegovaného režimu?*

Trap-and-emulate

Základní princip virtualizace

- **Citlivé instrukce:** Mění *globální stav* hostitele nebo se chovají rozdílně v závislosti na *globálním stavu*.
 - Příklad: Instrukce CLI (zákaz přerušení) mění globální stav CPU.
 - Chování instrukce SYSENTER závisí na globálním stavu – přepne procesor do privilegovaného módu a skočí na **vstupní bod jádra OS** (každý hostovaný OS má jiný vstupní bod).
- **Privilegované instrukce:** Pokus o jejich vykonání v uživatelském módu způsobí výjimku (**trap**), která je předána do privilegovaného módu k obslužení

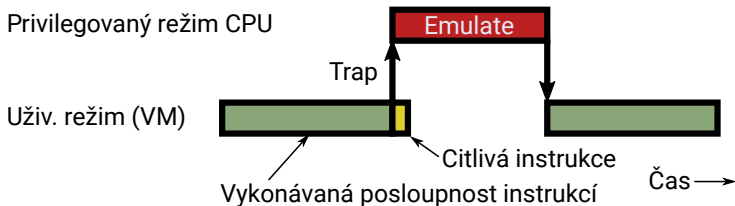
Trap-and-emulate

Základní princip virtualizace

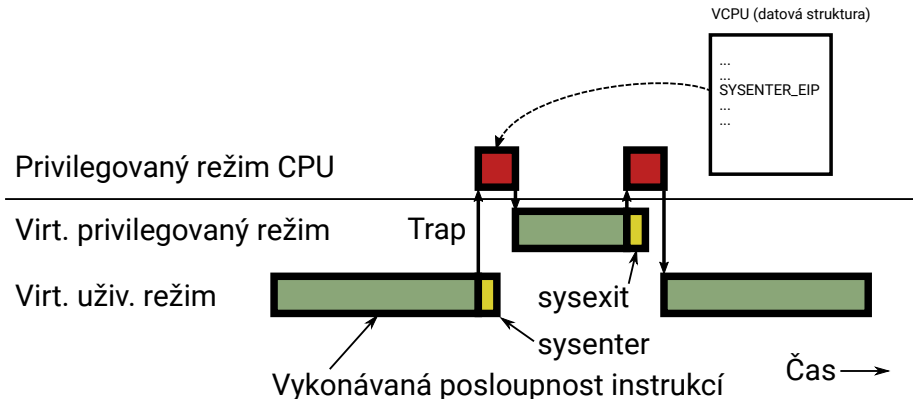
- **Citlivé instrukce:** Mění *globální stav* hostitele nebo se chovají rozdílně v závislosti na *globálním stavu*.
 - Příklad: Instrukce CLI (zákaz přerušení) mění globální stav CPU.
 - Chování instrukce SYSENTER závisí na globálním stavu – přepne procesor do privilegovaného módu a skočí na **vstupní bod jádra OS** (každý hostovaný OS má jiný vstupní bod).
- **Privilegované instrukce:** Pokus o jejich vykonání v uživatelském módu způsobí výjimku (**trap**), která je předána do privilegovaného módu k obslužení

Popek a Goldberg: Požadavky na virtualizovatelnost architektury CPU

„Všechny *citlivé* instrukce musí být zároveň *privilegované* instrukce.“



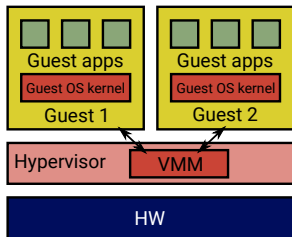
Virtualizace systémového volání



- Uživatelský kód běží stejně rychle jako bez virtualizace
- Přejechy do jádra a zpět jsou pomalejší kvůli emulaci

Rozdíl mezi hypervizorem a VMM

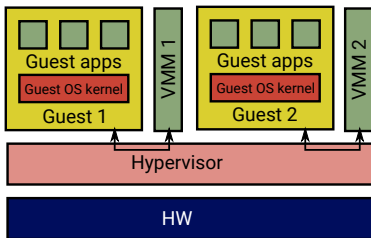
- **Hypervisor** – privilegovaný kód obsluhující výjimky generované v uživatelském režimu
- **Virtual machine monitor** – kód emulující virtuální HW (např. chování instrukce sysenter na předchozí stránce)



- Mnoho virtualizačních řešení slučuje funkci hypervizoru a VMM
- Příklad: VMware ESX, ...

Rozdíl mezi hypervizorem a VMM

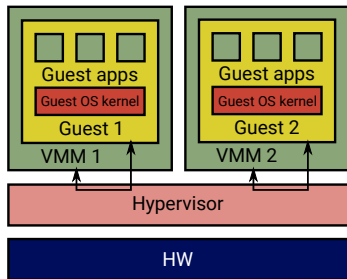
- **Hypervisor** – privilegovaný kód obsluhující výjimky generované v uživatelském režimu
- **Virtual machine monitor** – kód emulující virtuální HW (např. chování instrukce sysenter na předchozí stránce)



- Kód emulující HW je velký a složitý (zejména pro x86). Z bezpečnostního hlediska je rozumné nepouštět VMM v privilegovaném režimu.
- Hypervisor pouze odchytí výjimku a přepoše ji procesu VMM
- Příklady: NOVA, ...

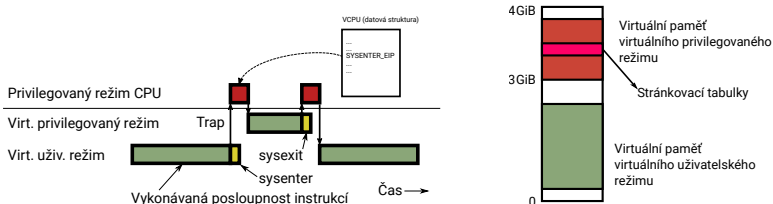
Rozdíl mezi hypervizorem a VMM

- **Hypervisor** – privilegovaný kód obsluhující výjimky generované v uživatelském režimu
- **Virtual machine monitor** – kód emulující virtuální HW (např. chování instrukce sysenter na předchozí stránce)



- VMM a hostovaný systém často tvoří jeden proces.
- Většinou stejně bezpečné jako předchozí případ
- Příklad: KVM + Qemu, viz také příklad dále

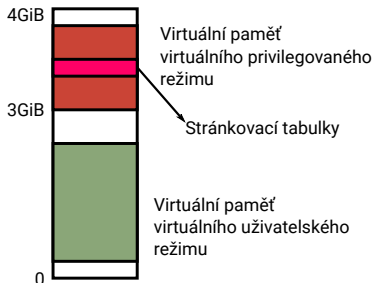
Virtualizace jednotky správy paměti (MMU)



- Základ bezpečnosti OS je, že kód běžící v uživatelském režimu nemůže modifikovat paměť jádra
- Virtuální uživatelský a privilegovaný režim ale běží ve skutečném uživatelském režimu a tudíž mají oba stejná oprávnění.
- VMM musí emulovat jednotku správy paměti
 - Při běhu virtuálního uživatelského režimu VMM nastaví CPU, aby používalo stránkovací tabulku, kde je povolen přístup pouze k virtuální uživatelské paměti.
 - Při běhu virtuálního privilegovaného režimu musí být přístup i do paměti hostovaného jádra – jiná stránkovací tabulka.
 - Jádro hostovaného OS nemůže mít přístup ke skutečné stránkovací tabulce.
 - Jak se dá řešit virtualizace přístupu ke stránkovacím tabulkám (např. `Ptab::insert_mapping()` v OS NOVA)?

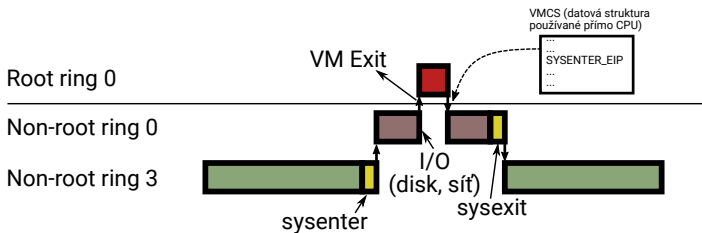
Virtualizace stránkových tabulek

- Trap-and-emulate & stínové stránkové tabulky
- Hypervizor nedá hostovanému jádru přístup pro zápis do paměti, kde jsou uloženy virtuální (tzv. stínové) stránkové tabulky
- Při pokusu o zápis do stránkových tabulek (Ptab::insert_mapping) dojde k výjimce.
- VMM se podívá, jak chtěl hostovaný OS stránku nastavit
- VMM zkontroluje, jestli host nepodvádí, nedělá chybu (izolace) a pak upraví skutečnou stránkovací tabulku

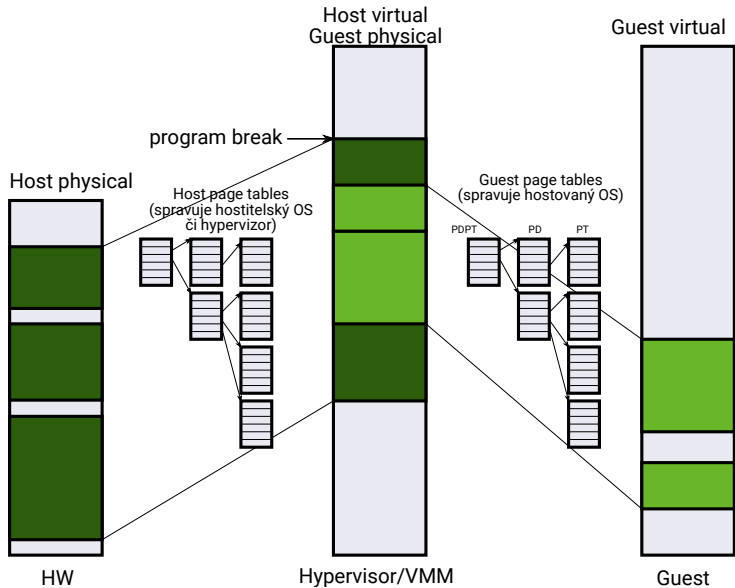


Hardwarově asistovaná virtualizace

- Implementace Trap-and-emulate je v mnoha případech pomalá
- Moderní CPU implementují to, co typicky dělá hypervisor, přímo v HW
- Intel: VT-x
 - Zavádí nový mód procesoru: **non-root execution**
 - V tomto módu jsou všechny citlivé instrukce zároveň privilegované (v původním root módu to neplatí; pushf, ...)
 - Existují tedy módy: Root ring 0–3, non-root ring 0–3
 - Hypervisor/VMM může nakonfigurovat které instrukce způsobí výjimky a přechod z non-root do root módu (tzv. VM Exit).
 - Instrukce sysenter (přechod z ring 3 do ring 0) je možné vykonat v non-root módu bez VM Exitu



Dvourstvé stránkování



Dvouúrovňové stránkování

- VT-x zavádí druhou úroveň stránkování
- O jedny stránkovací tabulky se stará hostitelský OS/hypervizor
- Hostovaný OS má své stránkovací tabulky
- Překlad virtuální adresy hosta na fyzickou adresu je pomalejší
- Ale je výrazně rychlejší než SW řešení pomocí stínových stránkovacích tabulek a trap-and-emulate.

Virtualizace vstupu a výstupu

- S většinou moderního HW se komunikuje pomocí:
 - **čtení/zápisu do registrů** (většinou mapovaných do paměti)
 - **datových struktur v paměti**
 - *Příklad (síťové rozhraní – Ethernet, zjednodušeno):* Do paměti uložím data, která chci odeslat a do registru síťového řadiče uložím adresu a délku dat. Na základě zápisu do registrů začne řadič připravená data odesílat.

Virtualizace vstupu a výstupu

- S většinou moderního HW se komunikuje pomocí:
 - **čtení/zápisu do registrů** (většinou mapovaných do paměti)
 - **datových struktur v paměti**
 - *Příklad (síťové rozhraní – Ethernet, zjednodušeno):* Do paměti uložím data, která chci odeslat a do registru síťového řadiče uložím adresu a délku dat. Na základě zápisu do registrů začne řadič připravená data odesílat.
- Virtualizaci paměti už máme vyřešenou
- Registry emulujeme pomocí trap-and-emulate
 - Hypervizor nastaví stránky, odpovídající registrům virtuálního HW jako „not present“
 - Každý pokus hostovaného OS o přístup k registrům způsobí výjimku (výpadek stránky, page fault)
 - Hypervizor výjimku obslouží posláním adresy (a zapisovaných dat) do VMM.
 - VMM zjistí o jaký registr se jedná a provede patřičnou akci – např. odeslání Ethernetového rámce přes skutečný HW.

Příklad: Wi-Fi rozhraní

```
$ lspci -v
```

```
02:00.0 Network controller: Intel Corporation Centrino Advanced-N 6205 [Tay  
Subsystem: Intel Corporation Centrino Advanced-N 6205 AGN  
Flags: bus master, fast devsel, latency 0, IRQ 31  
Memory at f7d00000 (64-bit, non-prefetchable) [size=8K]  
Capabilities: [c8] Power Management version 3  
Capabilities: [d0] MSI: Enable+ Count=1/1 Maskable- 64bit+  
Capabilities: [e0] Express Endpoint, MSI 00  
Capabilities: [100] Advanced Error Reporting  
Kernel driver in use: iwlwifi
```

Příklad: Wi-Fi rozhraní

```
$ lspci -v
```

```
02:00.0 Network controller: Intel Corporation Centrino Advanced-N 6205 [Tay
Subsystem: Intel Corporation Centrino Advanced-N 6205 AGN
Flags: bus master, fast devsel, latency 0, IRQ 31
Memory at f7d00000 (64-bit, non-prefetchable) [size=8K]
Capabilities: [c8] Power Management version 3
Capabilities: [d0] MSI: Enable+ Count=1/1 Maskable- 64bit+
Capabilities: [e0] Express Endpoint, MSI 00
Capabilities: [100] Advanced Error Reporting
Kernel driver in use: iwlwifi
```

Výpadek stránky v OS NOVA:

```
Ec::handle_exc Page Fault (eip=0x30f2 cr2=0x5000)
eax=0x5000 ebx=0x30a0 ecx=0x0 edx==0x0
esi=0x0 edi=0x4 ebp=0x1fe8 esp==0x1fb0
```

- Na jakou adresu se přistupovalo je uloženo v registru CPU CR2
- O jaký přístup šlo se zjistí podle instrukce na adrese EIP (mov \$123,0x5000)
- Při použití VT-x není nutné analyzovat instrukce – HW to udělá automaticky.

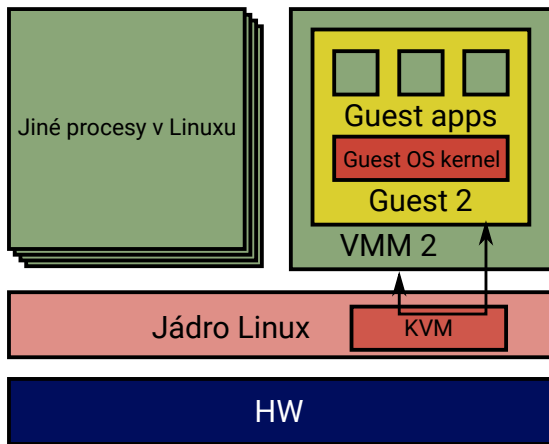
Příklad: KVM

- Hostovaný hypervizor, který je součástí Linuxového jádra
- Abstrahuje hardwarově asistovanou virtualizaci pomocí API (služeb ioctl)

Příklad: KVM

- Hostovaný hypervizor, který je součástí Linuxového jádra
- Abstrahuje hardwarově asistovanou virtualizaci pomocí API (služeb ioctl)
- Ukážeme si miniaturní VMM:
 - Nejjednodušší HW na virtualizaci: **sériový port**
zápis 1B do registru = požadavek na odeslání daného byte
 - 1 Nastavení paměti virtuálního stroje
 - 2 Načtení kódu do paměti VM
 - 3 Spuštění kódu ve VM
 - 4 Obsluha VM Exitů a emulace sériového portu
 - 5 Goto 3
- Viz také <https://lwn.net/Articles/658511/>

Příklad – Jednoduchý VMM pod Linuxem (KVM)



Příklad – Jednoduchý VMM pod Linuxem (KVM)

Inicializace

```
int kvm = CHECK(open("/dev/kvm", O_RDWR));
int vmfd = CHECK(ioctl(kvm, KVM_CREATE_VM, (unsigned long)0));

/* Allocate one aligned page of guest memory to hold the code. */
uint8_t *mem = CHECKPTR(aligned_alloc(0x1000, 0x1000));

/* Load the code to the guest memory */
int fd = open("hello.bin", O_RDONLY | O_CLOEXEC);
CHECK(read(fd, mem, 0x1000));
close(fd);

/* Map it to the second page frame (to avoid the real-mode IDT at 0). */
struct kvm_userspace_memory_region region = {
    .slot = 0,
    .guest_phys_addr = 0x1000,
    .memory_size = 0x1000,
    .userspace_addr = (uint64_t)mem,
};
CHECK(ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region));

int vcpufd = CHECK(ioctl(vmfd, KVM_CREATE_VCPU, (unsigned long)0));

/* Map the shared kvm_run structure and following data. */
size_t mmap_size = CHECK(ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, NULL));
struct kvm_run *run = CHECKPTR(mmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_SHARED, vcpufd, 0));

/* Initialize CS to point at 0, via a read-modify-write of sregs. */
struct kvm_sregs sregs;
CHECK(ioctl(vcpufd, KVM_GET_SREGS, &sregs));
sregs.cs.base = 0;
sregs.cs.selector = 0;
CHECK(ioctl(vcpufd, KVM_SET_SREGS, &sregs));
```

Příklad – Jednoduchý VMM v KVM

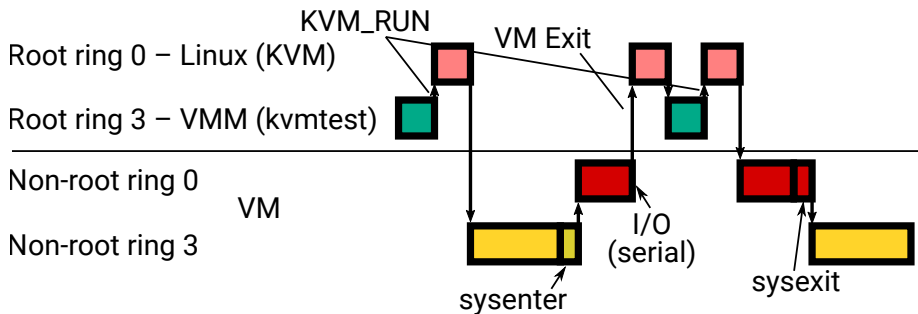
Běh VM a emulace sériového portu

```
/* Initialize registers: instruction pointer for our code and
 * initial flags required by x86 architecture. */
struct kvm_regs regs = {
    .rip = 0x1000,
    .rflags = 0x2,
};
CHECK(ioctl(vcpufd, KVM_SET_REGS, &regs));

/* Repeatedly run code and handle VM exits. */
while (1) {
    CHECK(ioctl(vcpufd, KVM_RUN, NULL));

    switch (run->exit_reason) {
    case KVM_EXIT_HLT:
        return 0;
    case KVM_EXIT_IO:
        if (run->io.direction == KVM_EXIT_IO_OUT && run->io.size == 1
            && run->io.port == 0x3f8 && run->io.count == 1)
            putchar(*(((char *)run) + run->io.data_offset));
        else
            errx(1, "unhandled KVM_EXIT_IO");
        break;
    default:
        errx(1, "exit_reason = 0x%x", run->exit_reason);
    }
}
```

Běh našeho VMM graficky



Obsah

1 Úvod

2 Virtualizace celého systému

- Virtualizace CPU
- Hardwarově asistovaná virtualizace
- Virtualizace vstupu a výstupu

3 Kontejnery

Kontejnery – Motivace

- Hardwarově-asistovaná virtualizace řeší problém náročnosti virtualizace na systémové zdroje jen částečně
 - Každý virtuální stroj potřebuje paměť pro jádro OS
 - Jádro OS má nějakou režii – procesorový čas (periodické přerušení od časovače apod.)
- Jediné, co chceme je, aby aplikace běžící v různých VM o sobě nevěděly
- Procesy OS implicitně sdílí mnohé informace
 - Tabulka procesů – každý proces si může zjistit jaké další procesy běží v systému
 - Identifikátory uživatelů – každý proces běží s právy nějakého uživatele a umí zjistit jací další uživatelé v systému jsou.
 - Souborový systém – proces často vidí, že existují soubory jiných uživatelů i když k nim nemá přístup
- VM tyto informace nesdílí

Kontejnery

- Řešení
 - Nepřidávat vrstvu abstrakce, ale
 - modifikovat OS, aby nebyly informace sdíleny mezi procesy
- Všechny „VM“ sdílí stejné jádro OS, ale jsou od sebe lépe izolovány.
- Linux:
 - Jmenné prostory (namespaces)
 - Řídící skupiny (cgroups)
- Jejich použití dohromady se říká **kontejner**

Jmenné prostory

- Místo toho, aby měl každý proces přístup ke „všemu“, je jeho jmenný prostor omezen podle pravidel nastavených administrátorem
- Typy jmenných prostorů v Linuxu:
 - Prostor identifikátorů procesů (PID)
 - Prostor identifikátorů uživatelů
 - Prostor připojených souborových systémů

Řídicí skupiny

- Řídí „spravedlivé“ přidělování systémových prostředků procesům
- Např. plánovač systému se snaží být spravedlivý ke všem vláknům
 - Pokud si jeden „kontejner“ vytvoří 10 000 vláken a druhý jen 10, bude první kontejner běžet 1000krát častěji než druhý
- Řešení: plánovač funguje hierarchicky
 - V první úrovni rozhoduje který kontejner poběží, v druhé úrovni které vlákno daného kontejneru.
 - Spravedlivost na všech úrovních.
- Řídicí skupiny:
 - Procesorový čas
 - Paměť
 - Síť
 - Disky
 - ...