

B4B350SY: Operační systémy

Virtualizace

Michal Sojka

`michal.sojka@cvut.cz`



10. prosince 2025

Obsah

1 Úvod

2 Virtualizace celého systému

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

3 Kontejnery

4 Závěr

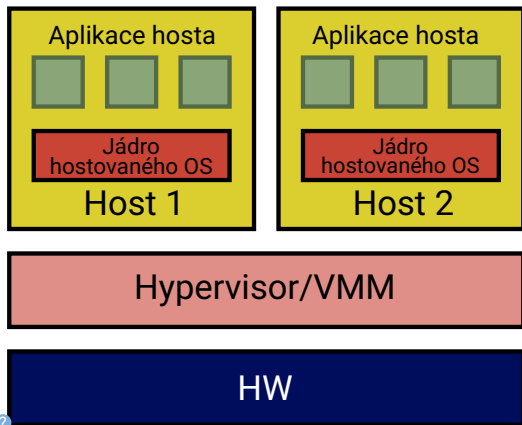
Co je virtualizace?

- 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

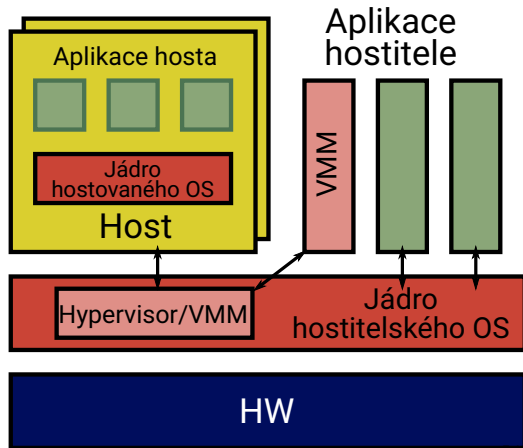
Typ 1 – nativní

- Příklady: VMware ESX, Citrix XenServer, NOVA



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
- Serverless computing
 - vyšší úroveň abstrakce než „klasický“ cloud computing
 - uživatel pouze napíše software, ale nestará se kde a jak poběží
 - správu virtuálních strojů, škálování apod. řeší automaticky poskytovatel cloudových služeb

Je virtualizace skutečně potřeba?

- Hlavní výhody virtualizace ze strany 6 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 spouštět 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)
 - $\Rightarrow$ 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 pomocí změn (uzavřených) OS

Možná řešení

- 1 Jedné vrstvy abstrakce se zbavíme jejím přesunem do hardwaru: HW akcelerace **virtualizace** $\Rightarrow$ lepší HW (viz dále)
- 2 Lepší architektura OS – **mikrojádro** (začínají se prosazovat v mnoha aplikacích, zejména pokud jde o bezpečnost) nebo aspoň **kontejnery**

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

Virtualizace CPU

- Klasické CPU vykonává kód ve dvou režimech (módech):
 - **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
 - virtuální uživatelský i virtuální privilegovaný režim běží ve **skutečném uživatelském režimu** procesoru (sdílí ho)
- **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?*

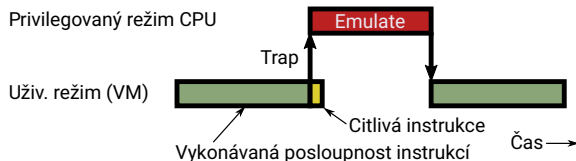
Trap-and-emulate

Základní princip virtualizace

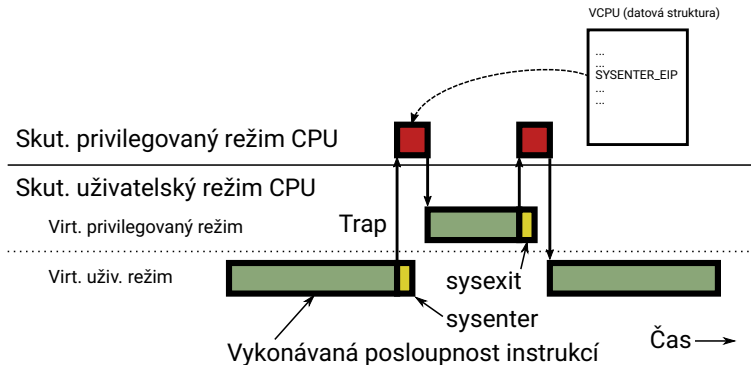
Popek a Goldberg: Požadavky na virtualizovatelnost architektury CPU

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

- **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** (ale 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



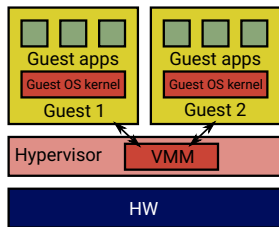
Virtualizace systémového volání



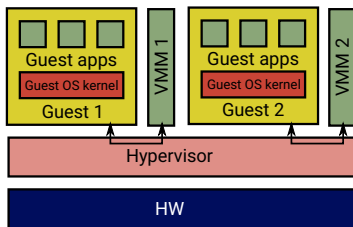
- Uživatelský kód běží stejně rychle jako bez virtualizace
- Přechody do jádra a zpět jsou pomalejší kvůli emulaci
 - Hypervizor/VMM si přečte virtualizovaný registr SYSENTER_EIP (položka v datové struktuře) a skočí zpátky do uživatelského režimu na přečtenou adresu.

Rozdíl mezi hypervizorem a VMM

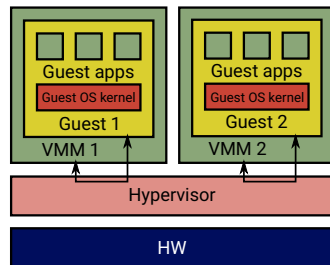
- **Hypervisor** – privilegovaný kód obsluhující výjimky generované v uživatelském režimu
- **Virtual machine monitor** – kód emulující virtuální HW (např. chování instrukce sysenter na předchozí stránce)



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

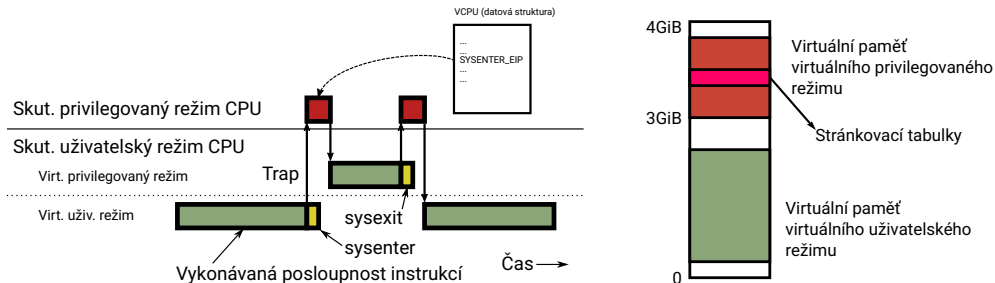


- Kód emulující HW je velký a složitý (zejména pro x86). Z bezpečnostního hlediska je rozumné nepouštět VMM v privilegovaném režimu.
- Hypervisor pouze odchytí výjimku a přepoše ji procesu VMM
- Příklady: NOVA, ...



- 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)

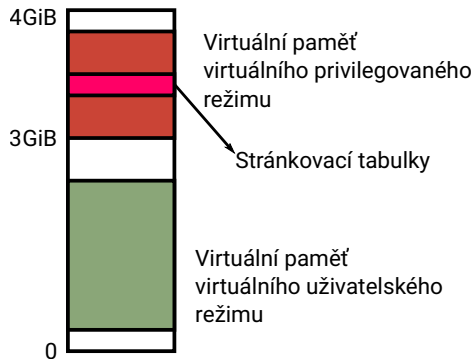


- Základ bezpečnosti OS je, že kód běžící v uživatelském režimu nemůže modifikovat paměť jádra
- Virtuální uživatelský a virtuální privilegovaný režim ale běží ve skutečném uživatelském režimu a tudíž mají oba stejná oprávnění.
- VMM musí emulovat jednotku správy paměti (MMU), aby zajistil izolaci paměťových prostorů: viz →
- 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

Princip: Trap-and-emulate & stínové stránkovací tabulky

- Hypervizor zpřístupní hostovanému jádru paměť, kde jsou uloženy virtuální (tzv. stínové) stránkovací tabulky, pouze pro čtení
- Při pokusu o zápis do stránkových tabulek (`Ptab::insert_mapping`) dojde k výjimce (trap)
- VMM obslouží výjimku tak, že se podívá, jak chtěl hostovaný OS stránku nastavit a zkontroluje, jestli host nepodvádí, nedělá chybu (izolace) atd. Pokud je vše v pořádku, upraví skutečnou stránkovací tabulku (emulate)



Hardwarově asistovaná virtualizace

■ Implementace

Trap-and-emulate je v mnoha případech pomalá

■ Moderní CPU

implementují to, co tradičně dělal hypervizor, přímo v HW

■ Intel: VT-x

■ Zavádí nový mód procesoru: **non-root execution**

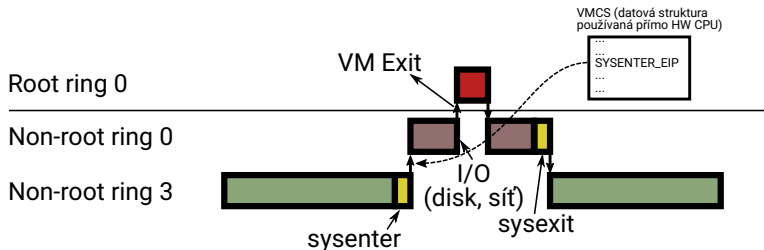
■ V tomto módu jsou všechny citlivé instrukce zároveň privilegované (v původním (tzv. *root*) módu to neplatí; viz např. instrukce `pushf`, ...)

■ Existují tedy módy: Root ring 0–3, non-root ring 0–3

■ Hypervizor/VMM může nakonfigurovat, které instrukce způsobí výjimky a přechod z non-root do root módu (tzv. VM Exit).

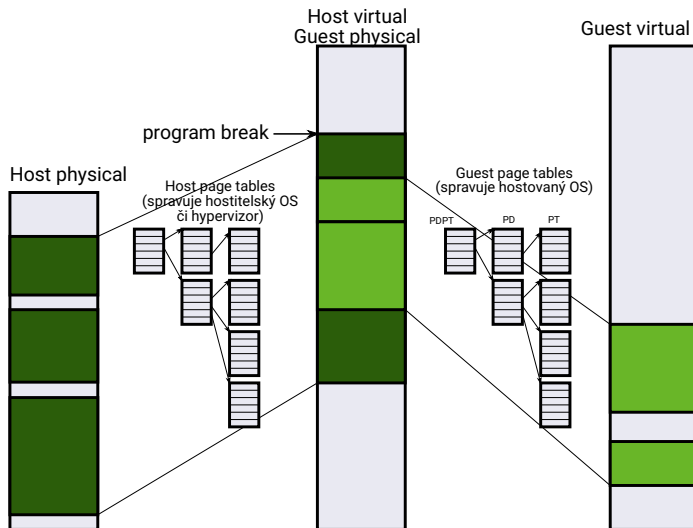
■ Instrukce `sysenter` (přechod z ring 3 do ring 0) je možné vykonat v non-root módu bez VM Exitu

■ Trap-and-emulate se ale stále používá pro emulaci I/O



Dvouvrstvé stránkování

Odstraňuje nutnost používání stínových stránkovacích tabulek



Dvouvrstvé stránkování

- VT-x zavádí druhou vrstvu 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, protože je potřeba udělat dva převody:
 - 1 Guest virtual → Guest physical (tj. Host Virtual)
 - 2 Host virtual → Host physical
- Změny tabulek jsou ale 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 rozhraní uložím adresu a délku dat. Na základě zápisu do registrů začne rozhraní 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 [Taylor Peak] (rev 34)
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:

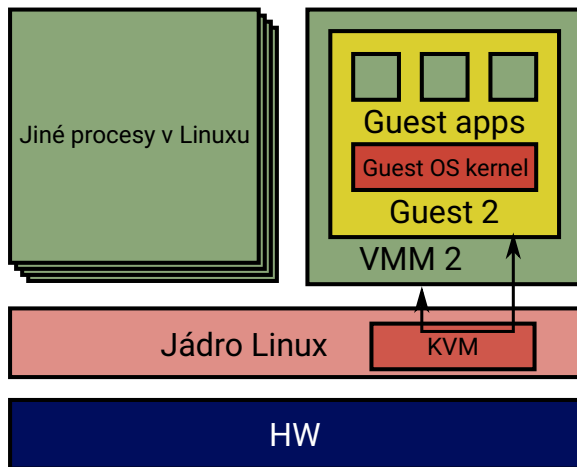
```
Ec::handle_exc Page Fault (eip=0x30f2 cr2=0xf7d00008)
eax=0x5000 ebx=0x30a0 ecx=0x0 edx==0x0
esi=0x0 edi=0x4 ebp=0x1fe8 esp==0x1fb0
```

- Na jakou adresu se přistupovalo je uloženo v registru CPU CR2
- O jaký přístup šlo se zjistí podle instrukce na adrese EIP (mov \$123,0xf7d00008)
- 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 `ioct1`)
- 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)

```
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));

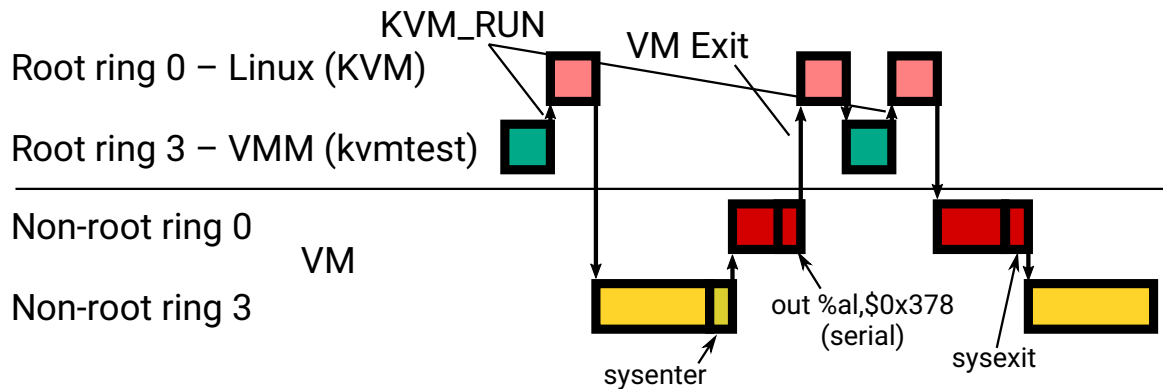
/* Initialize CS to point at 0, via a read-modify-write of sregs. */
struct kvm_sregs sregs;
CHECK(ioctl(vcpufd, KVM_GET_SREGS, &sregs));
sregs.cs.base = 0;
sregs.cs.selector = 0;
CHECK(ioctl(vcpufd, KVM_SET_SREGS, &sregs));
```

```
/* 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));

while (1) {
    /* Run the VM code (VM enter) */
    CHECK(ioctl(vcpufd, KVM_RUN, NULL));

    /* Handle VM exits */
    switch (run->exit_reason) {
        case KVM_EXIT_HLT:
            return 0;
        case KVM_EXIT_IO:
            /* Emulace sériového portu
             * 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 at port %d", run->io.port);
             * break;
             * default:
             *     errx(1, "exit_reason = 0x%x", run->exit_reason);
             */
    }
}
```

Běh našeho VMM graficky



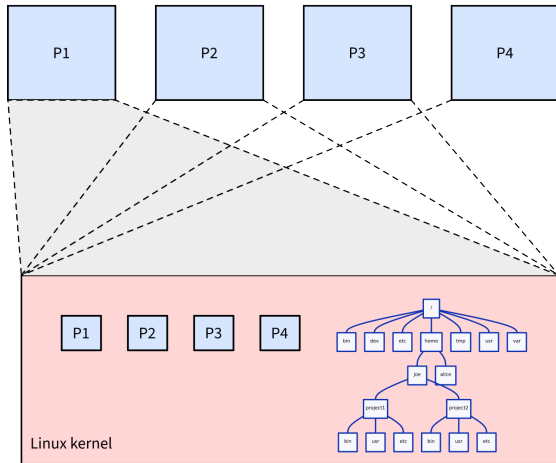
Kontejnery – Motivace

- 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.)
- Často chceme pouze, 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

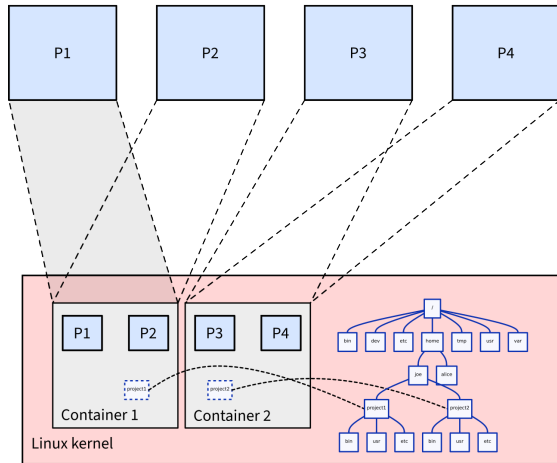
Bez kontejnerů:

Processes



S kontejnery:

Processes



Kontejnery

- Řešení
 - Nepřidávat vrstvu abstrakce (virtualizaci), ale
 - modifikovat jádro OS, aby nebyly informace sdíleny mezi procesy
- Všechny kontejnery sdílí stejné jádro OS, ale jsou od sebe lépe izolovány.
- Linux k tomu poskytuje následující mechanismy:
 - Jmenné prostory (**namespaces**) – řeší, aby proces „neviděl“ celý systém, ale jen část
 - Řídicí skupiny (**cgroups**) – omezují přidělování zdrojů (CPU, paměť) procesům
- Jejich použití dohromady se říká **kontejner**
- Systémové volání *clone*:
 - *fork()* je dnes implementován pomocí *clone()*
 - *pthread_create()* je také implementován pomocí *clone()*
 - vytvoření „kontejneru“ je jen volání *clone()* s jinými parametry

Vytvoření „kontejneru“

```
#define CHK(cmd) ({ int ret = (cmd); if (ret == -1) { perror(#cmd); exit(1); }; ret; })
#define STACK_SIZE (1024 * 1024)
char child_stack[STACK_SIZE];

int child_fn(void *arg) {
    CHK(sethostname("container", 9));
    CHK(chroot("./rootfs"));    // nastav kořenový adresář
    CHK(chdir("/"));
    CHK(mount("proc", "/proc", "proc", 0, ""));

    char *const args[] = {"/bin/sh", NULL};
    char *const env[] = {"PATH=/bin", NULL};
    return CHK(execvpe(args[0], args, env)); /* spust shell */
}

void main() {
    int flags = CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWIPC | SIGCHLD;
    pid_t container_pid = CHK(clone(child_fn, child_stack + STACK_SIZE, flags, NULL));
    printf("Container PID: %d\n", container_pid);
    waitpid(container_pid, NULL, 0);
}
```

Nástroje pro konfiguraci a správu kontejnerů

Docker, Podman, systemd-nspawn, LXD, ...

- Všechny tyto nástroje jsou „jen“ obálka systémového volání *clone()*
- Umožňují snadné vytváření kořenových adresářů
- Umožňují snadnou konfiguraci *cgroup* a nastavení sítě uvnitř kontejneru
- Např. soubory *Dockerfile* kombinují konfiguraci výše uvedeného do jednoho souboru

Příklad Dockerfile

```
FROM python:3.13
WORKDIR /usr/local/app

# Install the application dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy in the source code
COPY src ./src
EXPOSE 8080

# Setup an app user so the container doesn't run as the root user
RUN useradd app
USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Jmenné prostory

Namespaces

- Místo toho, aby měl každý proces přístup ke „všemu“, je jeho jmenný prostor omezen podle pravidel nastavených administrátorem
- Když se vytváří nový proces voláním `clone` (jako `fork`, ale nabízí víc možností), je možné specifikovat, že nový proces bude spuštěn v novém jmenném prostoru
- Typy jmenných prostorů v Linuxu:
 - **Prostor identifikátorů procesů (PID)** – proces nevidí procesy v jiných jmenných prostorech
 - **Prostor identifikátorů uživatelů** – proces má např. „rootovská“ práva v kontejneru, ale ne v celém systému
 - **Prostor připojených souborových systémů** – proces vidí jen omezenou část souborového systému
 - **Prostor síťových rozhraní** – proces vidí jen podmnožinu síťových rozhraní (nebo jen virtuální síťové rozhraní). Dá se tak omezit zda/s kým bude kontejner komunikovat po síti.

Řídicí skupiny

cgroups

- 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 (omezení na množství alokované paměti)
 - Síť (šířka pásma, latence)
 - Disky (šířka pásma, latence)
 - ...

Kvíz

Mechanismus trap-and-emulate se ve virtualizaci používá:

- A) pouze na architekturách ARM a MIPS.
- B) pouze na architektuře x86 u procesorů vyvinutých před r. 2005.
- C) pro emulaci I/O zařízení (síťová rozhraní, disky).
- D) pro emulaci vykonávání všech instrukcí virtualizovaného CPU.