Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Martin Děcký

Mechanismy virtualizace běhu operačních systémů

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
Studijní program: informatika, softwarové systémy

2006

Děkuji vedoucímu RNDr. Jakubu Yaghobovi, Ph.D, za cenné připomínky k této práci a také všem kolegům a přátelům za podporu při jejím vypracování.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 1. 8. 2006 Martin Děcký

Název práce: Mechanismy virtualizace běhu operačních systémů
Autor: Martin Děcký
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
E-mail vedoucího: `Jakub.Yaghob@mff.cuni.cz`

Abstrakt: Přehled a taxonomie technologií, které se používají pro virtualizaci běhu operačních systémů (spouštění více operačních systémů nebo více oddělených kontextů operačního systému současně na jednom počítači), jejich porovnání co se týče rychlosti, bezpečnosti, determinismu, míry izolace, accountingu, podporovaných platforem, emulace hardwaru apod. Inherentní problémy virtualizace (závislost na platformě, SMP, virtuální hardware). Emulace, simulace, virtuální stroje, paravirtualizace, partitioning. Přehled běžně dostupných virtualizačních produktů (Bochs, QEMU, Simics, VMware, Virtual PC, Virtual Server, OpenVZ, Denali, Mac on Linux, PearPC, Plex86, Xen, TRANGO, UML, Linux VServer, FreeBSD Jails, Solaris Zones) a jejich srovnání. Hardwarová podpora virtualizace (Power Hypervisor, V86, Vanderpool, Pacifica) a její využití.
Praktická demonstrace soft-partitioningu (rozdělení operačního systému na samostatné kontexty na úrovni kernelu) na kernelu SPARTAN.

Klíčová slova: Operační systémy, emulace, virtualizace, paravirtualizace, partitioning

Title: Mechanisms of Virtualizing Operating Systems Execution
Author: Martin Děcký
Department: Department of Software Engineering
Supervisor: RNDr. Jakub Yaghob, Ph.D.
Supervisor's e-mail address: `Jakub.Yaghob@mff.cuni.cz`

Abstract: Overview and taxonomy of the technologies used for virtualizing the execution of operating systems (running multiple operating systems or multiple separated operating system contexts simultaneously on a single computer), comparsion of the speed, security, determinism, level of isolation, accounting posibilities, supported platforms, level of hardware emulation, etc. Inherent problems of virtualization (platform dependency, SMP, virtual hardware). Emulation, simulation, virtual machines, paravirtualization, partitioning. Overview of commonly available virtualization products (Bochs, QEMU, Simics, VMware, Virtual PC, Virtual Server, OpenVZ, Denali, Mac on Linux, PearPC, Plex86, Xen, TRANGO, UML, Linux VServer, FreeBSD Jails, Solaris Zones) and their comparsion. Hardware assisted virtualization (Power Hypervisor, V86, Vanderpool, Pacifica) and their usage.
Practical demonstration of soft-partitioning (parcelling of the operating system into separate kernel contexts) on SPARTAN kernel.

Keywords: Oparating systems, emulation, virtualization, paravirtualization, partitioning

Kapitola 1

Úvod

Základním cílem textu je především vytvořit ucelený přehled technologií používaných pro virtualizaci běhu operačních systémů, s využitím společné terminologie zavést taxonomii těchto technologií, podat jejich obecnou charakteristiku, popsat jejich vlastnosti, výhody a nevýhody a na základě těchto pozorování shrnout inherentní problémy, které s sebou virtualizace přináší.

Další část diplomové práce se zaměřuje na konkrétní produkty implementující popsané technologie a na jejich specifické vlastnosti. V každé z hlavních kategorií je detailně popsán jeden produkt a je doplněn stručnější přehled produktů dalších. V závěru tohoto přehledu následuje porovnání těchto produktů z hlediska různých kritérií. Jedna kapitola práce je také věnována popisu několika hardwarových technologií sloužících pro podporu virtualizace.

V závěrečné části textu je zdokumentován postup rozšíření operačního systému o podporu virtualizace. Jako modelový příklad je použit experimentální operační systém HelenOS, vyvíjený v rámci MFF UK.

Diplomová práce navazuje na dlouhodobý zájem autora o problematiku operačních systémů a návrh rozhraní pro správu hardwaru na jedné straně a podporu aplikačních programů na straně druhé. Jejím přínosem by měl být komplexní pohled na problematiku virtualizace – teoretickými principy a taxonomií počínaje, přes přehled konkrétních produktů, hardwarových technologií a jejich srovnání a konče ukázkou praktické implementace podpory virtualizace.

1.1 Motivace výběru tématu

Podobně jako některé další zajímavé myšlenky uplatňované v informatice není ani virtualizace běhu operačních systémů žhavou novinkou a svou podstatou se jedná spíše o zobecnění již dávno používaných postupů než o zcela novátorskou ideu. Její využití v praxi bylo však dlouhou dobu výsadou high-end systémů a až postupně s rozvojem a zdokonalováním běžných osobních počítačů začala pronikat do oblasti běžných serverů a nakonec i desktopů.

Toto zpřístupnění a masové rozšiřování virtualizace má za důsledek dramatickou akceleraci vývoje v této oblasti a objevuje se velká diverzita různých řešení. Proto je dle mínění autora této diplomové práce vhodné aktuální stav zmapovat a také prakticky demonstrovat využití technologií virtualizace operačních systémů. Aktuálnost a důležitost tématu dokazují také poslední kroky významných IT společností jako Intel, AMD, Microsoft a další.

1.2 Struktura textu

Podrobnější komentář ke struktuře diplomové práce a obsahu jednotlivých kapitol:

Kapitola 2 Představení virtualizace, výhody, historický a aktuální vývoj, definice základních pojmů.

Kapitola 3 Taxonomie podle fyzického rozhraní a podle metody virtualizace, vlastnosti simulace, emulace, úplné virtualizace (nutné a dostačující podmínky, příklady jejich splnění na konkrétních platformách), virtuálních strojů, paravirtualizace a partitioningu.

Kapitola 4 Popis inherentních problémů, které s sebou metody virtualizace přináší (závislost na platformě, záležitosti víceprocesorových systémů).

Kapitola 5 Přehled simulátorů, dynamický překlad v simulátoru QEMU.

Kapitola 6 Přehled úplných virtualizérů, virtualizace hardwaru ve virtualizéru VMware.

Kapitola 7 Přehled paravirtualizérů, detailní popis vlastností paravirtualizéru Xen.

Kapitola 8 Přehled implementací partitioningu, detailní popis vlastností implementace Linux VServer.

Kapitola 9 Popis hardwarových technologií pro usnadnění virtualizace (VM86, Power Hypervisor, Vanderpool, Pacifica).

Kapitola 10 Srovnání metod i konkrétních implementací virtualizace.

Kapitola 11 Praktická demonstrace implementace partitioningu v systému HelenOS, portování systému HelenOS na virtuální platformu paravirtualizéru Xen.

Kapitola 12 Závěrečné shrnutí.

Kapitola 2

Virtualizace

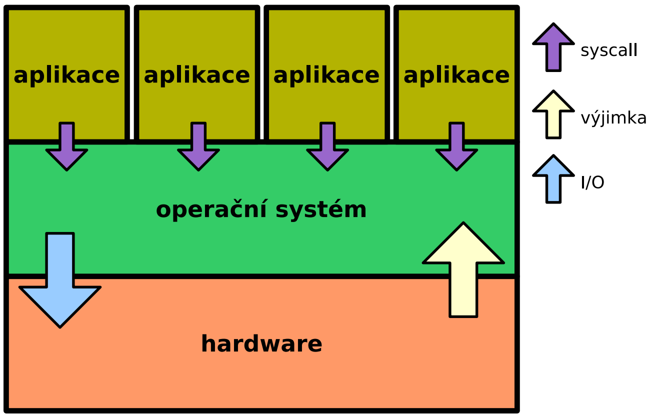
Návrh veškeré počítačové infrastruktury, kterou dnes používáme, by se s drobnou mírou zjednodušení dal shrnout do principu vytváření stále abstraktnějších rozhraní nad rozhraními jednoduššími. Mikroprocesor je tvořen základními polovodičovými prvky, které samy o sobě jsou charakterizovány nejlépe svými fyzikálními vlastnostmi. Tyto prvky jsou uspořádány do různých hradel, klopných a dalších obvodů, které již realizují na základě dobře definovaných úrovní elektrického napětí jednoduché logické operace. Důmyslné uspořádání těchto obvodů doplněné například o řízení na základě časového signálu umožňuje, aby mikroprocesor zpracovával program reprezentovaný posloupností instrukcí ve strojovém kódu, který sice může být poměrně jednoduchý, ale každá jeho instrukce abstraktně reprezentuje celou řadu komplexních propojení obvodů a hradel. Tento strojový kód, obsahující v zásadě jen jednoduché příkazy pro přesun dat, aritmetické a logické operace a podmíněné skoky, je možno generovat ze zdrojového kódu ve vyšším programovacím jazyce, kde lze reprezentovat mnohem abstraktnější operace jako cykly, volání podprogramů, objektovou reprezentaci kódu a dat atd. Ani zde však vzdalování od původního fyzikálního principu polovodiče, který v konečném důsledku vše realizuje, nekončí. Programátor nemusí ve zdrojovém kódu popisovat každý krok implementovaného algoritmu, ale může použít již existující knihovny (a tedy jen „spojuje“ abstraktní komponenty programu) nebo pomocí deklarativních jazyků již vůbec explicitně nepopisovat způsob výpočtu, ale pouze abstraktně formulovat řešený problém.

V počítačovém systému obvykle není jen jedna lineární posloupnost postupně se zvyšující abstrakce, ale můžeme vytvořit celý orientovaný graf reprezentující abstrahování, tedy implementaci obecnějších metod či rozhraní s využitím jednodušších již existujících prostředků. Tento text se bude především zabývat touto funkcí z hlediska operačního systému – vytvářením virtuálního stroje nad strojem fyzickým.

Nejobvyklejší je, že na jednom počítači běží v daný okamžik jen jeden operační systém, schématicky lze záležitost zachytit na obrázku 2.1. Virtualizace běhu operačního systému představuje tedy vsunutí další abstraktní vrstvy mezi hardware a operační systém tak, aby bylo možné současně spouštět více operačních systémů. Mechanismy virtualizace, jimiž se budeme v tomto textu zabývat speciálně, lze charakterizovat jako vytváření *virtuálního stroje* nad *strojem fyzickým*, nikoliv ovšem pro udržování běhového prostředí uživatelských programů, ale pro udržování běhového prostředí operačního systému.

2.1 Výhody virtualizace

Důležitou otázkou je, proč vlastně virtualizovat běh operačních systémů, když již samotný operační systém slouží jako prostředek, který spravuje hardwarové zdroje počítače, izoluje od sebe jednotlivé uživatelské procesy apod.



Obr. 2.1: Schéma počítače s klasickým operačním systémem.

Na virtualizaci můžeme například hledět jako na prostředek, jak tuto existující izolaci posílit. Většina operačních systémů vychází z paradigmatu běžných uživatelů, kteří mají srovnatelná omezená privilegia, a jednoho superuživatele, který má naopak v rámci systému práva prakticky neomezená. Virtualizace operačního systému nám potom umožní zjemnit tuto strukturu (jinými slovy mít více superuživatelů na jednom fyzickém stroji) bez nutnosti výrazné změny dosavadního bezpečnostního modelu.

Od představy bezpečnosti je již jen krůček k myšlence efektivní správy zdrojů, kdy z celkových fyzicky dostupných zdrojů počítače přidělíme jednotlivým virtuálním strojům jen jistou část (případně můžeme omezit jejich maximální využití). Sdílení fyzických zdrojů také vede k jejich efektivnějšímu využití (ekonomičtější průměrné zatížení) a lepší škálovatelnosti.

Nezávislost jednotlivých virtuálních strojů a operačních systémů v nich běžících je možné využít pro účely vývoje a ladění nestabilních verzí jader, testování kompatibility aplikačních programů s různými verzemi operačních systémů bez nebezpečí ovlivnění produkční konfigurace, provozování starších aplikací v původním prostředí, vyžaduje-li to jejich spolehlivý běh. Jako další variace na tuto možnost je testování softwaru s virtuální hardwarovou konfigurací, kterou fyzicky nevlastníme.

Je-li *virtuální rozhraní* (rozhraní mezi virtualizační vrstvou a virtuálním strojem, resp. operačním systémem) dostatečně nezávislé na *fyzickém rozhraní* (rozhraní mezi virtualizační vrstvou a fyzickým strojem), můžeme uvažovat o možnosti atomického ukládání kompletního stavu virtuálního stroje a přesunu (migrace) tohoto stavu na jiný fyzický stroj. To umožňuje provádět změnu hardwarové konfigurace fyzického stroje bez nutnosti znatelného výpadku služeb, ale také slouží jako způsob zvýšení spolehlivosti služeb jako takových (v případě hardwarového selhání lze přesunout funkční stav virtuálního stroje na záložní fyzický stroj).

2.2 Historie

První použití virtualizačních metod se datuje do 70. let 20. století, kdy převažující technologií byly mainframy. Tyto i na dnešní dobu poměrně výkoné počítače měly velmi velké pořizovací a provozní náklady, proto bylo snahou jejich provoz maximálně zefektivnit. Provoz více operačních systémů na jednom fyzickém stroji umožnilo lépe se přizpůsobit potřebám různých uživatelů a to bez vzájemného ovlivnění jejich běhu.

Model 67 počítače IBM S/360 byl první navržen nejen s podporou oddělení aplikací v neprivilegovaném režimu, ale také s možností virtualizace privilegovaného režimu. Virtualizována byla obsluha přerušení, I/O operace i přístupy do paměti.

Operační systém VM/CMS počítačů IBM S/360 a jeho nástupců obsahoval virtualizér VM jako svou základní součást, zatímco běžný systém CMS byl jednouživatelský. Různí uživatelé systému VM/CMS tedy pracovali na celém jednom virtuálním stroji. Jednalo se o první kombinaci operačního systému a úplné virtualizace hardwaru. Návrh instrukční sady S/360 je z hlediska virtualizace natolik dobrý, že lze snadno provozovat virtualizaci i rekurzivně.

První virtualizační software pro běžné PC se objevil v 90. letech 20. století v podobě produktu VMware, který provádí úplnou virtualizaci, přestože se jedná na architektuře PC o velmi složitou záležitost (pro urychlení grafických operací používá volitelně možnost přímého volání některých svých funkcí z hostovaného operačního systému). Jako varianty virtualizace, které jsou efektivnější z hlediska celkové režie, ale vyžadují jisté úpravy operačních systémů, které virtualizují, se po roce 2000 objevují paravirtualizéry (např. Xen). Také různé metody partitioningu, které sice neumožňují současný běh více různých operačních systémů, ale dovolují rozdělit daný operační systém do několika nezávislých částí (kontextů, zón, partitions atd.) začínají být velice populární.

V posledních letech si také výrobci mikroprocesorů AMD a Intel uvědomují vzrůstající význam virtualizace a rozšiřují instrukční sady platforem IA-32 a AMD64 o možnosti, jak vyřešit dosavadní velkou pracnost a časovou režii úplné virtualizace.

2.3 Aktuální vývoj

Lze určitě říci, že rok 2006 je pro virtualizaci přelomový. Open source implementace partitioningu jako Linux VServer a OpenVZ si postupně nacházejí cestu do běžně používaných distribucí GNU/Linuxu (a začlenění nějaké univerzální formy partitioningu do standardního jádra Linuxu je již zřejmě také poměrně blízko), stejně běžný začíná být v distribucích také open source paravirtualizér Xen (ovšem také systémy jako FreeBSD a NetBSD jej aktivně podporují, na podpoře v OpenSolarisu se již také pracuje).

Pravděpodobně pod sílícím tlakem na používání těchto volně dostupných virtualizačních nástrojů uvolnila společnost VMware některé konkrétní produkty své softwarové řady pro úplnou virtualizaci IA-32 k volnému používání (zatímco nejnáročnější high-end varianty jsou stále komerční) a klíčový hráč softwarového trhu, společnost Microsoft, reagovala podobně u nedávno získaných produktů Virtual PC a Virtual Server. Implementace lepší podpory virtualizace v procesorech společností Intel a AMD (technologie Vanderpool a Pacifica) je jen dalším důkazem nastoleného trendu.

Nasazení nějaké formy virtualizace se brzo stane stejně běžné jako je nyní používání operačních systémů s ochranou paměti. Umožní to efektivnější využití hardwarových zdrojů, snazší administraci, vyšší dostupnost služeb a jejich větší bezpečnost. V další fázi asi dojde k prohloubení doposud spíše okrajových vlastností virtualizace jako migrace živých systémů apod.

2.4 Jiné významy virtualizace

Tento text se především zabývá virtualizací v již nastíněném významu – vytváření vrstvy mezi fyzickým a virtuálním rozhraním, která umožňuje rozdělit prostředky jednoto fyzického stroje mezi obecně více strojů virtuálních.

V obecném významu může ovšem virtualizace znamenat také spojení více fyzických prostředků do jednoho prostředku virtuálního (zřejmý případ jsou disková pole). Proto zmiňme na tomto místě alespoň velmi stručně dva další případy virtualizace operačních systémů, kterými se v tomto textu již nadále zabývat nebudeme.

Paralelní virtuální stroj (Parallel Virtual Machine) je knihovna, která umožňuje provádět distribuovaný výpočet na více běžných počítačích spojených sítí. Pro algoritmus upravený s použitím knihovny PVM se tato množina počítačů jeví jako jeden virtuální víceprocesorový počítač.

Grid je mechanismus na vytváření virtuálního počítače pro distribuované výpočty, který využívá výpočetní výkon, paměť a další zdroje více nezávislých počítačů.

2.5 Základní pojmy

V celém následujícím textu budeme předpokládat význam následujících několika základních pojmů, jejichž vzájemný vztah názorně demonstrují obrázky 2.2 a 2.3.

Fyzické rozhraní Rozhraní mezi fyzickým strojem a virtualizační vrstvou.

Fyzický stroj Výpočetní systém, který je realizovaný pomocí klasických elektronických nebo mechanických součástek.

Hypervisor Společné označení pro *virtualizér* a *paravirtualizér*.

Hypervolání Explicitní žádost kódu běžícího ve virtuálním stroji o spuštění funkce hypervisoru. Analogie systémového volání v případě aplikačního programu a jádra operačního systému.

Monitor virtuálního stroje (Virtual Machine Monitor) V kontextu tohoto textu synonymum pro *virtualizér*.

Operační systém Softwarový systém sestávající obvykle z *jádra* (běžícího v privilegovaném režimu procesoru), knihovních funkcí a aplikačních programů (běžících v uživatelském režimu procesoru) vytvářející běhové prostředí pro další aplikační programy.

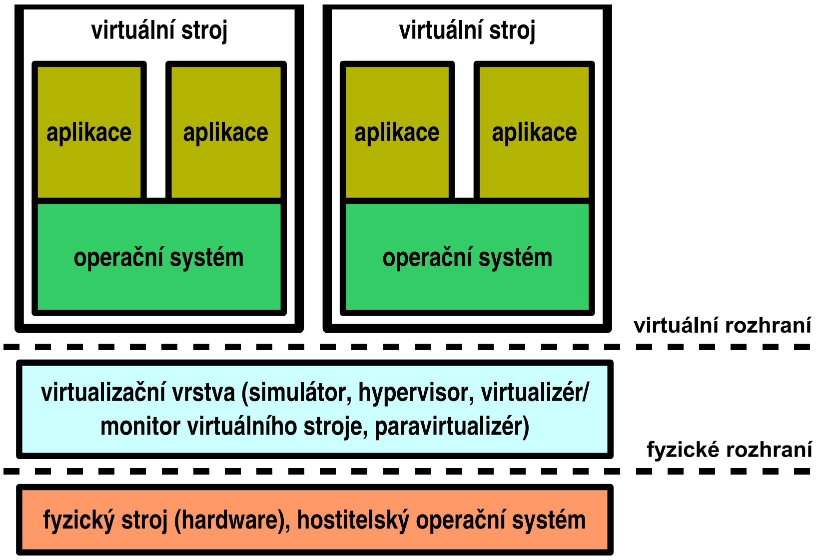
Paravirtualizér Program řídící paravirtualizaci.

Virtualizér Program řídící úplnou virtualizaci.

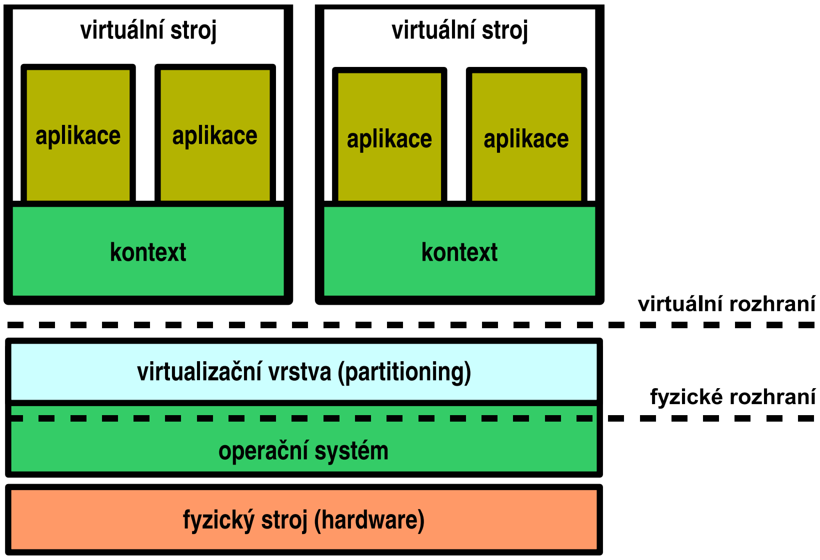
Virtualizační vrstva Obecné označení pro *simulátor*, *virtualizér*, *paravirtualizér* a subsystém realizující partitioning.

Virtuální rozhraní Rozhraní mezi virtualizační vrstvou a virtuálním strojem.

Virtuální stroj Izolovaný výpočetní systém, který poskytuje všechny prostředky pro samostatný běh programů. V případě simulace a virtualizace představuje virtuální stroj víceméně přesnou repliku stroje fyzického.



Obr. 2.2: Vzájemná souvislost jednotlivých pojmů v případě simulace, úplné virtualizace a paravirtualizace.



Obr. 2.3: Vzájemná souvislost jednotlivých pojmů v případě partitioningu. Situace je komplikovanější v tom, že virtualizace pomocí partitioningu je integrální součástí jádra operačního systému, které je takto rozděleno na část obsahující fyzické rozhraní a část obsahující virtuální rozhraní.

Kapitola 3

Taxonomie a vlastnosti

Na metody virtualizace lze hledět různými hledisky, mezi nejpodstatnější patří asi způsob implementace samotné virtualizační vrstvy (která silně ovlivňuje virtuální rozhraní) a umístění fyzické vrstvy (zda komunikuje přímo s hardwarem nebo je závislá na hostitelském operačním systému).

3.1 Fyzické rozhraní

Tato taxonomie se týká simulace, úplné virtualizace a paravirtualizace. V případě partitioningu nemá smysl, protože u něj je fyzické rozhraní vždy součástí operačního systému, který rozděluje na jednotlivé kontexty.

3.1.1 Přímé rozhraní s hardwarem

Virtualizační vrstva může svým fyzickým rozhraním komunikovat přímo s fyzickým strojem, tedy hardwarovými prostředky. Tento případ bývá častý v případě úplné virtualizace a paravirtualizace, naopak poměrně vzácný u simulace.

Výhody tohoto přístupu jsou především v minimální režii a nejsnadnějším způsobu kontroly hardwarových prostředků, naopak znamenají komplikaci v tom, že hypervisor musí v takovém případě sám zprostředkovat správu hardwarových prostředků – nejen jejich přidělování virtuálním strojům, ale také inicializaci a řízení (v praxi často obsahuje vnitřní vrstvu ovladačů zařízení jako by se jednalo o operační systém).

3.1.2 Rozhraní s hostitelským operačním systémem

Tato varianta bývá typická u simulátorů a také velmi častá v případě virtualizérů a paravirtualizérů. Inicializace a řízení hardwarových prostředků je na starosti hostitelského operačního systému, simulátor nebo hypervisor v něm potom běží jako téměř obyčejný aplikační program (u hypervisorů je často nutné rozšířit operační systém o možnost řízení některých privilegovaných stavů tímto aplikačním programem) a virtualizuje hardwarové prostředky již zprostředkované aplikačním rozhraním operačního systému.

Výhoda tohoto přístupu spočívá ve snadnější podpoře různého hardwaru z hlediska simulátoru/hypervisoru (o tu se stará hostitelský operační systém), nevýhoda je poněkud větší režie způsobená dalším rozhraním navíc a občas nebezpečí konfliktů plynoucích z toho, že hypervisor musí být schopen ovlivňovat některé privilegované stavy).

3.1.3 Hybridní rozhraní

Kombinace obou výše uvedených přístupů se často používá u paravirtualizérů. Samotný paravirtualizér sice běží přímo na hardwaru, ale sám inicializuje a řídí jen hardware nezbytně nutný ke svému běhu. O další periférie se stará operační systém v jednom privilegovaném virtuálním stroji, který má přístup k omezeným prostředkům stroje fyzického a zároveň jejich funkce zprostředkovává dalším virtuálním strojům.

3.2 Metoda virtualizace

Jednotlivé virtualizační metody se liší především ve způsobu realizace virtualizační vrstvy a přímo ovlivňují podobu virtuálního rozhraní.

3.2.1 Simulace

Podobně, jako lze za pomocí počítačů věrně simulovat různé fyzikální děje v kinetice, dynamice, optice nebo elektronice, je principiálně možné simulovat chování jednoho počítačového systému pomocí jiného. Nezajímáme-li se zrovna o průběhy elektrických jevů, nemusíme ovšem v tomto případě simulovat skutečně celý fyzický počítač, ale pouze vnitřní rozhraní hardwaru vůči programu – jinými slovy instrukční sadu a chování procesoru, přístup do paměti, komunikační protokoly a chování sběrnic, vstupně/výstupních periferií atd.

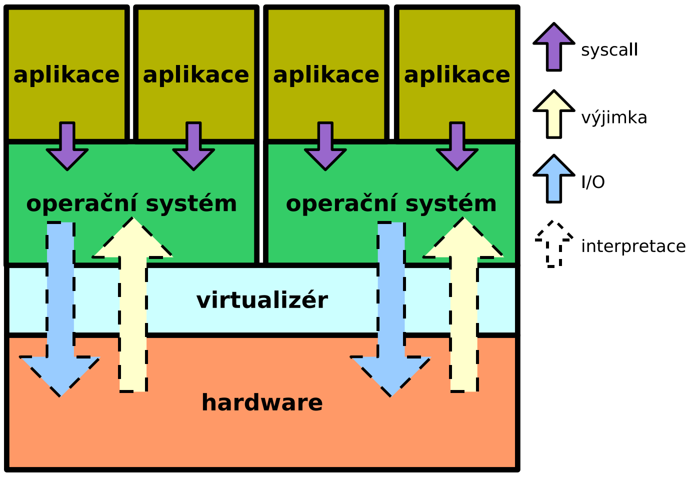
Simulátor je software běžící typicky jako aplikační program v rámci hostitelského operačního systému, který interpretuje strojový kód podle specifikace hostovaného počítačového systému. Veškerý simulovaný hardware existuje pouze virtuálně v rámci simulátoru (jeho vnitřní stavy představují podmnožinu vnitřních stavů simulátoru), zatímco data virtuálních vstupně/výstupních periferií jsou mapována (obvykle pomocí prostředků hostitelského operačního systému) na soubory, jiné programy nebo skutečný hardware.

Simulace se snaží v ideálním případě dodržovat vlastnost *ekvivalence*, kdy program běžící v rámci simulovaného počítače pracuje stejně jako kdyby běžel na počítači reálném, který by byl vybaven hardwarem ekvivalentním s hardwarem simulovaným. Do požadavků ekvivalence se obvykle ze zřejmých důvodů nezahrnuje otázka časování (konkrétně rychlost nebo propustnost simulovaného systému) a otázka deterministického chování simulátoru.

Simulace procesoru

Instrukční sada, registry a chování procesoru patří k základním součástem každého simulátoru (samostatné simulátory procesoru bývají obvykle nazývány *emulátory*). Definitorické vymezení pojmu simulátor obvykle zahrnuje implicitní podmínku, že simulátor procesoru každou simulovanou instrukci interpretuje a nikoliv provádí přímo na fyzickém procesoru (přestože jsou obě instrukční sady shodné nebo kompatibilní).

Naivní interpretace znamená, že každá instrukce virtuálního stroje je dekódována a jedná-li se o instrukci korektní a přípustnou, je zavolána funkce simulátoru, která ji odsimuluje (tedy změní vnitřní stav simulátoru podle specifikace instrukce). Simulátory používající tento způsob interpretace mají největší režii, ale přesto se tato metoda používá, protože umožňuje simulátor snadno naprogramovat a udržovat. Výhoda je také v přenositelnosti (na jiné fyzické rozhraní, resp. hostitelský operační systém) a univerzálnosti (často hovoříme o multisimulátorech, které dovedou simulovat celou řadu virtuálních rozhraní a na prakticky libovolném fyzickém rozhraní).



Obr. 3.1: Schéma úplné virtualizace.

Dynamický překlad se používá v případě, že je potřeba dosáhnout menší režie při simulaci. Snaží se přeložit co největší posloupnosti instrukcí virtuálního stroje do odpovídajících posloupností instrukcí fyzického stroje a takto přeložený kód používat pokud možno opakovaně.

Simulace periférií

V rámci vlastnosti ekvivalence je potřeba, aby simulátor velmi věrně napodoboval skutečné chování hardwarových periférií simulované platformy do nejmenších detailů, což obvykle představuje chování sběrnic, jednotlivých hardwarových obvodů a časování hardwarových událostí.

3.2.2 Emulace

Emulace je metoda svým principem velmi blízká simulaci, mezi těmito dvěma pojmy existuje jen velmi neostrá hranice. Obvykle se uvádí, že emulace se nesnaží vytvářet virtuální stroj jako dokonalou repliku stroje fyzického, ale simuluje fyzický stroj ve velmi zjednodušené podobě (jen některé prostředky, případně ne zcela věrně) nebo jen převádí jedno rozhraní na jiné.

Příkladem emulátorů v podobě zjednodušených simulátorů jsou například interprety, které dovedou vykonávat instrukce instrukční sady daného procesoru a emulují přístupy do paměti vybavováním údajů z vlastního adresního prostoru, ale nesimulují žádné konkrétní periferie fyzického stroje vybaveného daným procesorem. Pro vstup a výstup se obvykle používá velmi zjednodušený model, kdy pevně určené místo v emulované paměti slouží pro výstup znaků na obrazovku (každý zapsaný znak emulátor vypíše na svůj vlastní aplikační výstup) a podobně pro čtení znaků z virtuální klávesnice se používá jiné místo v paměti (volitelně může emulátor generovat virtuální přerušení při příchodu nového znaku).

Emulátory ve smyslu převádění jednoho rozhraní na jiné (například soubor systémových volání jednoho operačního systému na systémová volání jiného operačního systému) stojí již zcela mimo rámec tohoto textu.

3.2.3 Úplná virtualizace

Mechanismus úplné virtualizace[1] je v mnoha ohledech velmi podobný simulaci, ovšem s tím podstatným rozdílem, že provádění některých instrukcí není nutné simulovat, ale mohou se provádět nativně. Čím více instrukcí je možno provádět bez simulace, tím efektivněji virtualizovaný kód běží.

Úplná virtualizace není na rozdíl od simulace možná na libovolném fyzickém rozhraní (a zároveň libovolném virtuálním rozhraní, neboť tyto spolu v této situaci velmi úzce souvisí). Existuje soubor nutných a dostačujících podmínek úplné virtualizace.

Dostačující podmínky úplné virtualizace

Jak uvádějí Popek a Goldberg v dnes již klasickém článku [1], dostačující podmínky pro implementaci úplné virtualizace umožňují určit, zda lze vytvořit pro dané fyzické rozhraní virtualizér, který je schopen vytvořit virtuální rozhraní bezpečně virtualizující veškeré dostupné zdroje (procesory, paměť a vstupně/výstupní periferie).

Virtuální stroj vytvořený virtualizérem musí být charakterizován těmito vlastnostmi:

Ekvivalence Program běžící v rámci virtuálního stroje by měl fungovat naprosto stejně jako kdyby běžel přímo na fyzickém rozhraní bez přítomnosti virtualizéru. Mezi podmínky ekvivalence se přitom obvykle nepočítá zachování přesného časování, chování při nedostupnosti nějakého zdroje a externí příčiny jednotlivých událostí, které nemůže program žádnými prostředky detekovat.

Správa zdrojů Virtualizér by měl zcela kontrolovat veškeré virtualizované zdroje. To znamená, že program běžící v rámci virtuálního stroje by neměl mít možnost jakkoliv ovlivnit zdroje fyzického stroje jiným způsobem než zprostředkovaně přes virtualizér, neměl by mít možnost používat jiné zdroje než explicitně povolené a v případě potřeby by měl mít virtualizér možnost přidělené zdroje opět odebrat.

Aby virtuální stroj splňoval tyto podmínky, musí být platforma buď dokonale simulována, nebo v případě virtualizace musí hardware poskytovat takové nástroje, které virtualizéru umožní splnit je. Jedná se především o strukturu instrukční sady procesoru, paměťového modelu, zpracování výjimek a přerušení a přístup k periferiím.

Instrukční sada

Procesor, který umožňuje plnou virtualizaci, musí být schopen běžet ve dvou režimech – privilegovaném a uživatelském. Instrukce instrukční sady jsou poté rozděleny do těchto tří množin:

- **Privilegované instrukce**
  Vyvolávají výjimku, pokud jsou spuštěny v uživatelském režimu, a nevyvolávají výjimku, pokud jsou spuštěny v privilegovaném režimu.
- **Instrukce ovlivňující stav**
  Instrukce mění stav zdrojů systému (paměťový model, stav periferií).
- **Instrukce závislé na stavu**
  Chování těchto instrukcí závisí na stavu zdrojů systému.

Věta: Virtuální stroj může být zkonstruován, pokud množina instrukcí ovlivňujících stav a množina instrukcí závislých na stavu jsou podmnožinami množiny privilegovaných instrukcí.

[1] Přívlastek *úplná* používáme z toho důvodu, abychom jasně odlišili tuto konkrétní metodu virtualizace od obecného pojmu.

Pokud jsou všechny instrukce ovlivňující stav privilegovanými instrukcemi, je jejich provádění v uživatelském režimu procesoru vždy detekovatelné virtualizérem, čímž je zajištěna správa všech zdrojů (kód prováděný ve virtuálním stroji nemůže přímo ovlivnit fyzické zdroje). Analogicky možnost detekce a ošetření všech instrukcí závisejících na stavu zdrojů umožňuje hypervisoru dosáhnout ekvivalence s fyzickým strojem (veškeré přístupy k virtualizovaným zdrojům jsou hypervisorem převedeny na příslušné přístupy k příslušným fyzickým zdrojům).

Současně umožňuje hardware splňující toto tvrzení návrh efektivní virtualizace, kdy je potřeba simulovat pouze privilegované instrukce, zatímco ostatní instrukce je možno provádět nativně, aniž by hrozilo porušení některé důležité vlastnosti.

Pokud daná platforma obsahuje množinu *kritických instrukcí* (instrukcí ovlivňujících stav nebo závislých na stavu, které nejsou privilegované), může být implementace virtualizace stále možná například za pomoci dynamické rekompilace, kdy jsou kritické instrukce při načítání nového kódu do paměti virtuálního stroje přepisovány vhodnými privilegovanými instrukcemi, které při zpracování umožní virtualizéru provézt ošetření původní instrukce.

Architektura IA-32 obsahuje několik instrukcí, které narušují vlastnost *ekvivalence* a proto je potřeba použít složité prostředky k dosažení úplné virtualizace:

- SGDT, SIDT, SLDT
  Tyto instrukce po řadě ukládají hodnotu registru GDTR, registru IDTR, resp. hodnotu selektoru z LDTR (do paměti nebo v posledním případě také do obecného registru). Tyto hodnoty představují stav fyzického stroje, nikoliv stav stroje virtuálního (jedná se o prostředky paměťového modelu, který musí virtualizér virtualizovat), ovšem při vyvolání výše zmíněných instrukcí v uživatelském režimu nedojde k vyvolání žádné výjimky a program ve virtuálním stroji tedy přečte hodnoty, podle níž může poznat, že neběží na fyzickém stroji.
- SMSW, PUSHF
  První instrukce umožňuje uložit do obecného registru hodnotu řídicího registru CR0, druhá ukládá na zásobník hodnotu registru příznaků. Podobně jako instrukce první skupiny nevyvolávají v uživatelském režimu výjimku. Některé bity registru CR0 a registru příznaků ovšem opět odrážejí stav fyzického stroje, který nemusí být shodný se strojem virtuálním (chráněný režim, příznaky FPU, příznak přepnutí úlohy apod.)
- LAR, LSL, VERR, VERW
  Instrukce slouží k přečtení přístupových práv segmentů, limitu segmentu a provedení testu na možnost čtení, resp. zápisu do segmentu. V závislosti na vzájemné kombinaci stavů fyzického a virtuálního stroje mohou tyto instrukce, které opět nevyvolávají výjimku, vracet z hlediska virtuálního stroje neočekávané hodnoty.
- PUSH
  Tato instrukce pro práci se zásobníkem umožňují ukládat také segmentové registry, které ovšem nesou informaci o režimu procesoru, pro který jsou dané segmenty přístupné. Operační systém běžící v rámci virtuálního stroje předpokládá, že běží v privilegovaném režimu procesoru, ovšem hodnoty selektorů musí odpovídat segmentům přístupným z uživatelského režimu. Tím opět dochází k potížím analogickým s těmi uvedenými výše.
- STR
  Podobně jako v několika předchozích případech umožňuje tato instrukce, která čte segment registru úlohy, operačnímu systému běžícímu ve virtuálním stroji detekovat, že segment není určen pro privilegovaný režim procesoru, jak se očekává, ale pro uživatelský režim.
- MOV
  Některé varianty této instrukce umožňují přečíst hodnotu segmentových registrů opět bez toho, aby v uživatelském režimu došlo k výjimce. Analogicky jako výše to umožňuje detekovat nečekané hodnoty selektorů.

Paměťový model

Nutnou podmínkou virtualizace paměti jako základního zdroje fyzického i virtuálního stroje je existence mechanismu správy paměti, který umožňuje definovat překlad paměťové adresy v rámci virtuálního stroje na obecně jinou paměťovou adresu v rámci stroje fyzického a navíc dovoluje omezit virtuální stroj na používání jen těch adres, které mu byly explicitně přiděleny.

Nejjednodušší takový paměťový model představuje bázovou adresu (která určuje posunutí fyzických adres vůči adresám virtuálního stroje) a limit (určující rozsah adres použitelných virtuálním strojem na $\langle 0, \mathrm{limit} \rangle$). Je zřejmé, že bázová adresa a limit vyjadřují stav paměti a proto je lze měnit jen instrukcemi ovlivňujícími stav.

Model přerušení a výjimek

Přerušení nebo výjimka, která vznikne současně s během virtuálního stroje, musí být nejprve prozkoumána a případně ošetřena virtualizérem. Tato podmínka je obvykle splněna už tím, že virtuální stroj běží v uživatelském režimu procesoru, zatímco přerušení a výjimky se zpracovávají v privilegovaném režimu. Virtuální stroj musí byt schopen měnit svou virtuální podobu vektoru přerušení (nebo jiného mechanismu, který se na dané platformě používá) a případně maskování jednotlivých přerušení bez ovlivnění fyzického stroje.

Přístup k periferiím

Veškerý přístup k periferiím musí být realizován výhradně pomocí mechanismu, který může virtualizér bezpečně odchytit, případně pomocí paměťově mapovaných oblastí, pokud daná fyzická platforma disponuje dostatečně sofistikovaným paměťovým modelem.

Nutné podmínky úplné virtualizace

Tři nutné podmínky úplné virtualizace jsou podle [4] tyto:

1. Způsob dekódování neprivilegovaných instrukcí musí být shodný se způsobem dekódování instrukcí privilegovaných.
2. Musí existovat mechanismus na ochranu paměti fyzického stroje a ostatních virtuálních strojů od aktuálně běžícího virtuálního stroje. Speciálně v případě, že fyzické rozhraní je realizováno hostitelským operačním systémem, tak musí existovat mechanismus na předání informace o výjimce vzniklé v rámci virtuálního stroje z hostitelského operačního systému do virtualizéru.
3. Virtualizér musí být schopen detekovat snahu virtuálního stroje provézt privilegovanou instrukci a musí být schopen její očekávaný efekt odsimulovat.

Nejrozšířenější architektura IA-32, kromě toho, že nesplňuje dostačující podmínky úplné virtualizace, také nedodržuje třetí nutnou podmínku virtualizace:

- POPF
  Tato běžná instrukce pro načtení registru příznaků ze zásobníku, nevyvolává v uživatelském režimu výjimku. Její chování se však zásadně liší podle toho, zda má být z hlediska virtuálního stroje provedena v privilegovaném nebo uživatelském režimu. Zatímco v privilegovaném režimu nastavuje celou sadu dostupných příznaků, v uživatelském režimu nastavení některých příznaků tiše ignoruje. Bez speciálního ošetření této instrukce nejen, že není kód virtuálního stroje prováděn ekvivalentně, ale je dokonce prováděn nekorektně.

- POP, CALL, JMP, INT, RET
  Všechny tyto instrukce při použití konkretních hodnot selektorů (v případě volání, skoku, přerušení a návratu často ve spojitosti s volacími bránami dochází ke změně zásobníků) brání ve virtualizaci tím, že jejich chování se liší podle toho, zda dochází ke změně režimu procesoru nebo ne. Tyto testy budou ovšem ve virtuálním stroji dopadat jinak, protože všechny segmenty jsou určeny pro uživatelský režim.
- MOV
  Problém nastává při pokusu o uložení hodnoty segmentového registru SS, kdy opět mechanismus ochrany paměti provádí testy, které dopadají jinak než při provádění stejného kódu na fyzickém stroji.

Řešení

Při implementaci virtualizérů na IA-32 se používají některé techniky (viz [5]), které dovolují obejít omezení uvedená v předchozích odstavcích.

Základním předpokladem je analýza proudu instrukcí před jejich spuštěním. Virtualizér si eviduje rámce paměti, které obsahují dosud neprověřené instrukce (tyto rámce nejsou ve virtuálním stroji mapovány), ověřené rámce jsou namapovány pouze s možností čtení, aby snaha o jejich modifikaci (samomodifikující se kód nebo pouhé spouštění nového kódu v rámci virtualizovaného operačního systému) vyvolaly výjimku a nově zapsaný kód mohl být opět analyzován.

Kritické a problémové instrukce jsou ošetřeny breakpointy, které způsobí explicitní vyvolání virtualizéru, a původní instrukce musí byt odsimulovány.

Instrukce skoku jsou také ošetřeny breakpointem, ale je navíc zapnuto krokování instrukcí tak, aby po skoku do stránky, která zatím nebyla analyzována, došlo k její analýze. Je-li skok pevný (nezávisí-li na operandech) a směřuje-li do již analyzované stránky, je možné jej nadále provádět bez simulace.

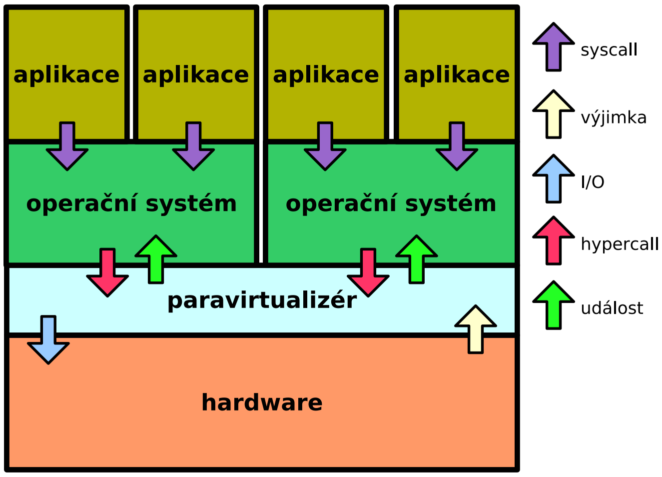
Jak ovšem optimálně implementovat breakpointy, o kterých hovoří předchozí odstavce? Použití hardwarových breakpointů není možné, protože jich je jen omezený počet a navíc často nebývají virtualizéru k dispozici. Pokud bychom přímo použili softwarové breakpointy (trap instrukce), musíme modifikovat kód prováděný ve virtuálním stroji, což umožní, aby tak program detekoval, že neběží na stroji fyzickém.

Jedno z možných řešení je to, že ve skutečnosti se bude modifikovaný kód provádět z jiného fyzického rámce, než z jakého k němu bude mít přístup pro datové čtení/zápis virtuální stroj. Zároveň je potřeba zajistit, aby také virtuální adresy (fyzické adresy z pohledu virtuálního stroje) vzájemně odpovídaly, čehož lze například dosáhnout použitím mechanismu segmentace (je-li k dispozici) nebo řízeným nastavením hodnot TLB cachí za předpokladu, že je fyzický procesor vybaven odděleným datovým a kódovým TLB.

3.2.4 Virtuální stroje

Termín *virtuální stroj* je podobně jako řada termínů v oblasti IT a informatiky silně přetížen, přestože většina jeho definic má podobné vlastnosti. Virtuální stroj z hlediska simulace, emulace a virtualizace představuje ekvivalent fyzického stroje, jehož rozdílné vlastnosti a chování nejsou detekovatelné uvnitř virtuálního stroje.

Existují však také stroje, které jsou virtuální i svým návrhem a nekopírují hardwarové rozhraní žádného fyzického stroje. Mezi nejznámější virtuální stroje tohoto druhu patří Java Virtual Machine, sestávající z interpretu p-kódu (zásobníkového assembleru nad instancemi objektů) a sady standardních knihoven. Z takto definovaného virtuálního stroje, který vlastně inherentně umožňuje



Obr. 3.2: Schéma paravirtualizace.

virtualizaci, protože interpretů p-kódu můžeme spustit několik, je naopak možné odvodit stroj fyzický (*Java Processor*), který bude hardwarově implementovat jeho rozhraní.

Virtuální stroje, které jsou definovány paravirtualizéry (viz následující oddíl), představují jakýsi přechod mezi oběma těmito krajními možnostmi – virtuální stroj jako ekvivalent stroje fyzického na jedné straně a virtuální stroj navržený zcela nezávisle na straně druhé. Takový virtuální stroj má obvykle neprivilegovanou instrukční sadu shodnou s instrukční sadou daného fyzického stroje, také z důvodu usnadnění portace bývá cílem maximální možnou podmnožinu rozhraní ovlivňujících stav zdrojů zachovat shodných, přesto však neumožňuje virtuální stroj provádět všechny privilegované instrukce.

3.2.5 Paravirtualizace

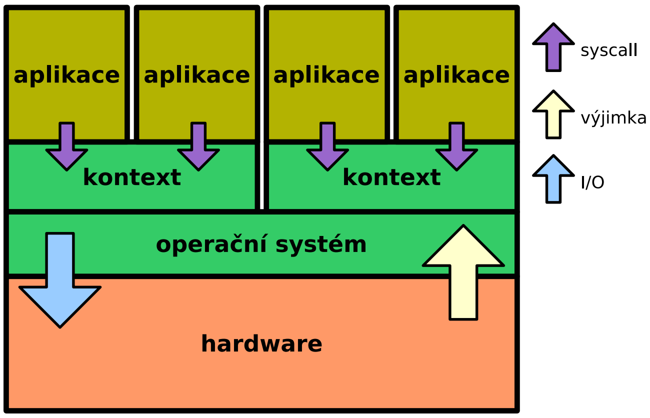
Na paravirtualizaci se lze pro první přiblížení dívat jako na virtualizaci, která nedovede sama o sobě dodržet některé podmínky nutné pro úplnou virtualizaci (viz strana 22, sekce 3.2.3) a naopak vyžaduje, aby virtualizovaný operační systém o přítomnosti hypervisoru věděl a explicitně používal jeho rozhraní pro změnu stavu virtuálního stroje a přístup k některým zdrojům. Rozhraní mezi hypervisorem a virtualizovaným operačním systémem tedy není „neviditelné“ v podobě rozhraní ekvivalentního s rozhraním mezi fyzickým strojem a operačním systémem.

Složitost vytvoření virtualizéru se v případě paravirtualizace tedy převádí na problém vhodné úpravy operačního systému, což ovšem bývá obvykle záležitost snadná, protože většina moderních operačních systémů je připravena na možnost běhu na různých hardwarových platformách.

Paravirtualizér by měl být vždy schopen dodržet druhou Popek-Goldbergovu podmínku (viz strana 20, sekce 3.2.3) a neumožnit virtuálnímu stroji, aby bez jeho vědomí manipuloval se zdroji fyzického stroje. Není-li možné ani tuto podmínku dodržet, potom paravirtualizace neposkytuje odolnost vůči chybám operačního systému běžícího ve virtuálním stroji a musí předpokládat jeho naprostou korektnost.

3.2.6 Partitioning

Mnohdy není potřeba, aby na fyzickém stroji běželo několik nezávislých operačních systémů, pouze je vhodné vytvořit z hlediska koncového uživatele „iluzi“ oddělených systémů (kontextů). Jádro operačního systému je společné pro všechny kontexty, ovšem ke každé entitě (vlákno, proces, adresový



Obr. 3.3: Bezpečnostní kontexty v rámci jednoho operačního systému.

prostor, soubor atd.) je připojen identifikátor kontextu, do kterého patří, a při každém pokusu o přístup k této entitě (ať už interně v rámci jádra nebo zprostředkovaně z uživatelské aplikace) se provádí test, zda entita náleží do správného kontextu. Obvykle se na základě kontextu určuje také pouhá viditelnost jednotlivých entit, takže uživatelská aplikace běžící v daném kontextu nemůže detekovat přítomnost aplikací a dalších objektů existujících v jiných kontextech.

Tento přístup je především výhodný v tom, že režie testů náležení do kontextu je nepatrná i ve srovnání s mechanismy paravirtualizace, navíc přidání těchto testů většinou nebývá složité a ve vhodně navrženém jádře operačního systému se může jednat jen o vhodné rozšíření existujícího bezpečnostního modelu.

Metoda se v některých implementacích nazývá *soft-partitioning*, aby se zdůraznilo, že se jedná o řešení na úrovni jádra operačního systému. Označení *hard-partitioning* se naopak někdy používá k označení metody, kdy hypervisor přiděluje virtuálním strojům sady procesorů víceprocesorového fyzického stroje.

Jak demonstruje i tento text v sekci 11.2, přidání základní podpory partitioningu do rozumně napsaných operačních systémů je velmi přímočarou a snadnou záležitostí, která spočívá v označení příslušnosti jednotlivých entit jádra do jednotlivých kontextů a vytvoření nemnoha testů na jejich povolenou interakci a viditelnost. V dobře navrženém jádře totiž mohou entity interagovat jen několika málo dobře definovanými způsoby.

Kapitola 4

Inherentní problémy virtualizace

Jak již bylo v úvodu tohoto textu naznačeno, všechny metody virtualizace představují jen definování rozhraní mezi hardwarem a programem. Výjimečné postavení mají ovšem v tom, že jejich cílem je „vklínit“ se mezi již existující rozhraní hardware - operační systém a to za předpokladu jeho minimálních změn. Míra těchto změn roste po řadě v posloupnosti simulace, úplná virtualizace, paravirtualizace, partitioning (emulaci a virtuální stroje, pro účely srovnání v této kapitole okrajové záležitosti, nebudeme uvažovat).

Některé problémy virtualizace jsou však principiálně nevyhnutelné. V konečném důsledku musí každý virtualizovaný systém (i v případě hypotetického použití rekurzivní virtualizace) běžet na reálném hardwaru, jehož zdroje jsou konečné. Nelze je tedy neomezeně virtualizovat a jejich přidělování je buď exkluzivní (obvykle části fyzické paměti) nebo pomocí spoolingu[1]) (obvykle u periférií).

4.1 Závislost na platformě

Pokud hovoříme o závislosti virtualizace na platformě, je vždy třeba rozlišovat mezi závislostí na straně fyzického rozhraní a na straně virtuálního rozhraní. Proberme jednotlivě každou metodu.

4.1.1 Simulace

Simulátor se už ze své definice snaží maximálně věrně simulovat chování fyzického stroje, tedy virtuální rozhraní je také maximálně závislé na platformě. Simulátor vytváří nezávisle na platformě, na které sám běží, virtuální prostředí konkrétního počítače včetně procesoru, paměti a periferních zařízení.

Co se týče platformové závislosti fyzického rozhraní, tam se situace výrazně liší podle konkrétní implementace simulátoru. Zabývejme se nejprve instrukční sadou. Virtuální instrukční sada může být interpretována tak, že každé instrukci odpovídá funkce (nebo jiná část) simulátoru, která je napsána ve vyšším programovacím jazyce. Tato naivní interpretace je sice nejpomalejší, ale zároveň přenositelnost takového simulátoru omezuje jen přenositelnost použitého vyššího programovacího jazyka.

Pokročilejší metody interpretace virtuálních instrukcí vyžadují obvykle silnější vazbu na fyzickou platformu. Při dynamickém překladu, kdy jsou celé bloky virtuálních instrukcí převedeny na posloupnost fyzických instrukcí a takto vzniklý kód je přímo spouštěn, je pochopitelně závislost na fyzické platformě maximální. Je ovšem možné zvolit méně krajní varianty, kdy výstupem dynamického překladu virtuálního kódu nejsou přímo fyzické instrukce, ale opět nějaký mezikód, který lze interpretovat platformě nezávislým způsobem.

[1] Každý zájemce získá virtuální abstrakci zařízení, se kterým může pracovat, ale jen některé požadavky jsou v daném okamžiku fyzicky realizovány. Ostatní čekají ve frontě.

Kromě instrukční sady ovšem simulátor simuluje také periférie, u nichž bývá nejproblematičtější napodobení vhodných časování. Z tohoto pohledu je obvykle celý simulátor navržen jako diskrétní simulátor, který si udržuje seznam dvojic (*čas*, *událost*), kde *čas* je absolutní počet virtuálních procesorových cyklů a *událost* je změna stavu nějaké virtuální periférie. Při vykonání každé virtuální instrukce je připočtena k čítači procesorových cyklů hodnota odpovídající definované délce provádění dané instrukce a jsou provedeny všechny události, jejichž časy jsou menší než hodnota čítače (navíc jsou také vyřazeny ze seznamu dvojic). Vykonání instrukce může mít za následek požadavek na budoucí změnu stavu periférií, tyto události jsou proto přidány s odpovídajícím časem v budoucnosti do seznamu dvojic.

4.1.2 Úplná virtualizace

Při použití úplné virtualizace závislost na straně fyzického i virtuálního rozhraní spolu velmi úzce souvisí. Protože neprivilegovaná instrukční sada virtuálního stroje se při úplné virtualizaci provádí přímo (a virtualizér obvykle nemá prostředky pro doplňkovou simulaci neprivilegovaných instrukcí), odpovídá také virtuální procesor procesoru fyzickému. Hypervisor potom obvykle sám implementuje úplnou virtualizaci jen pro některé konkrétní procesorové modely, takže vlastnosti jsou ještě zredukovány na nejbližší nižší kompatibilní model.

Již tedy ze samotného principu virtualizace není možné napsat virtualizér, který by byl svým fyzickým rozhraním nezávislý na platformě.

Virtualizace periférií není v případě virtualizace možná metodou diskrétní simulace, protože není možné po každé vykonané instrukci zjišťovat její přesnou dobu provádění. Virtuální periferie proto obvykle pracují v reálném čase a jejich časování je přibližně korelováno s virtuálním procesorovým časem.

4.1.3 Paravirtualizace

Z hlediska instrukční sady se paravirtualizace od virtualizace nijak výrazně neliší – neprivilegované instrukce se provádějí přímo a privilegované instrukce jsou zakázány a nahrazeny hypervoláním. Paravirtualizér musí ovšem tato hypervolání obsloužit obvykle velmi hardwarově specifickým způsobem bez možnosti velké abstrakce, takže fyzické a v důsledku toho také virtuální rozhraní jsou silně platformově závislé.

U virtuálních periférií se obvykle nepoužívá stejné rozhraní pro jejich řízení jako u fyzického stroje, ale paravirtualizér (nebo v některých případech privilegovaný virtuální stroj) poskytuje virtuálním strojům abstraktní rozhraní (často rozlišené podle druhu periférie), které funguje pouze na principu změny stavu a nevyžaduje dodržení přesného časování.

4.1.4 Partitioning

Partitioning je jediná virtualizační metoda, která může být implementována zcela hardwarově nezávisle. Je to způsobeno tím, že virtuální a fyzické rozhraní partitioningu není vklíněno mezi rozhraní hardware - operační systém, dokonce nemusí být ani mezi platformově závislými a platformově nezávislými částmi operačního systému, ale lze jej kompletně implementovat v platformově nezávislém kódu.

Přístup k hardwaru je v tomto případě zcela stejný jako v případě operačního systému bez paravirtualizace, pouze jsou implementována omezení na to, ke kterým konkrétním perifériím mohou jednotlivé kontexty přistupovat.

4.2 Víceprocesorové systémy

V celém předchozím textu jsme tiše předpokládali, že fyzický stroj je vybaven jedinou centrální výpočetní jednotkou (procesorem) a také virtuální stroj obsahuje jediný virtuální procesor. Symetricky víceprocesorové (SMP) systémy a jejich virtualizace přináší některá úskalí, se kterými se ne všechny metody virtualizace vyrovnávají stejně snadno.

4.2.1 Simulace

Koncepce většiny simulátorů je jednovláknová, mohou tedy efektivně využít jen jediný procesor fyzického stroje pro každý virtuální stroj. Simulace víceprocesorových virtuálních strojů se provádí deterministickým prokladem virtuálních instrukcí více procesorů.

Vícevláknové simulátory, kdy každé vlákno simuluje jeden nebo více procesorů, je pochopitelně možná, ale z důvodu složitosti takové simulace (především dodržení globálního časování) se obvykle nepoužívá.

4.2.2 Úplná virtualizace

Většina virtualizérů je navržena pro virtualizaci jediného procesoru, samotný proces může být ovšem rozdělen do více vláken (jedno provádí samotnou virtualizaci, jiné implementuje virtuální periferie), takže je zde jistý prostor pro vhodné využití více fyzických procesorů.

V poslední době byla uvedena spíše experimentální podpora vytváření víceprocesorových virtuálních strojů. Na rozdíl od simulace zde není požadavek na dodržení přesného deterministického časování, složitost a režii navíc přináší nutnost implementace sdílení a zamykání některých datových struktur virtualizéru týkajících se jednotlivých virtuálních procesorů a stavu celého systému.

4.2.3 Paravirtualizace

Paravirtualizéry byly od počátku navrhovány s podporou SMP. Každý fyzický procesor má svůj obraz jako virtuální procesor a ty mohou být libovolně (někdy i dynamicky za běhu) přidělovány jednotlivým virtuálním strojům.

Problém sdílení a zamykání datových struktur zde existuje stejný jako v případě úplné virtualizace, ovšem díky tomu, že virtuální rozhraní může být situaci vhodně přizpůsobeno, je tato záležitost srovnatelná se zamykáním struktur jádra víceprocesorového operačního systému.

4.2.4 Partitioning

Kvalita podpory víceprocesorových systémů u virtualizace pomocí partitioningu je srovnatelná s podporou samotného jádra operačního systému, proces virtualizace kontextů totiž v typickém případě nijak neovlivňuje plánování jednotlivých procesů nebo vláken na jednotlivé procesory. Ze všech metod virtualizace je zde granularita mapování virtuálních procesorů na fyzické nejmenší, ve své podstatě totiž nemůžeme hovořit o tom, že by existovaly nějaké ucelené virtuální procesory.

Pro administraci bývá ovšem občas vhodné omezit možnost plánování procesů jednotlivých kontextů na konkrétní fyzické procesory (nebo počty procesorů), čímž lze dosáhnout sice hrubého, ale velmi úsporného omezení maximálního výpočetního výkonu, které mohou jednotlivé virtuální stroje spotřebovat.

Kapitola 5

Přehled simulátorů

Tato kapitola představuje stručný přehled několika simulátorů. Pochopitelně se nejedná o vyčerpávající výčet, především v oblasti simulátorů starších 8bitových a 16bitových počítačů existuje nepřeberná řada projektů. Použitelnost simulátorů jako prostředků virtualizace je kvůli výrazné ztrátě výpočetního výkonu v praxi spíše omezená, mohou ovšem sloužit jako referenční platforma. Také tam, kde nelze dodržet dostačující podmínky úplné virtualizace, mohou se použít techniky simulace v omezené míře.

Zásadní spojitost potom existuje mezi simulátory a úplnými virtualizéry v případě virtualizace periférií.

5.1 QEMU

Univerzální simulátor QEMU vyvíjený Fabricem Bellardem je příkladem použití dynamického překladu instrukcí pro dosažení vysoké rychlosti simulace. Speciální modul *kqemu* pro jádro Linuxu navíc používá některé metody úplné virtualizace, takže umožňuje na fyzickém IA-32 stroji provádět většinu instrukcí neprivilegovaného režimu IA-32 a VM86 režimu přímo bez nutnosti emulace.

Kromě simulace celého systému umožňuje QEMU také pouze emulovat instrukční sadu procesoru a tak spouštět binární kód uživatelských aplikací v Linuxu na jiné platformě, než na které byly původně přeloženy (využívá se toho, že API systémových volání v Linuxu je na většině platforem shodné).

5.1.1 Struktura simulátoru

QEMU je rozděleno na několik relativně samostatných subsystémů. Základním subsystémem je emulátor procesoru, v současné době je podporována emulace IA-32, AMD64, MIPS R4000, SPARC, ARM a PowerPC. Další subsystém se stará o simulaci periferních zařízení (grafická karta, sériový port, klávesnice, myš, síťová karta, PCI sběrnice, IDE sběrnice atd.), jako samostatné moduly jsou implementovány bloková a znaková zařízení, která tvoří fyzické rozhraní pro tyto virtuální periférie. Samostatné subsystémy jsou také debugger a uživatelské rozhraní pro ovládání simulátoru. Definice jednotlivých simulovaných strojů (např. PC, Sun4m, PowerMac) určují iniciální konfiguraci simulovaných zařízení.

5.1.2 Přenositelný dynamický překlad

Ojedinělou vlastností QEMU je podpora dynamického překladu virtuálních instrukcí na instrukce fyzického stroje bez toho, aby bylo nutné psát fyzické ekvivalenty virtuálních instrukcí v assembleru a tudíž nepřenositelně.

Vstupními daty pro dynamický překlad jsou funkce v jazyce C interpretující každou instrukci virtuální instrukční sady. Tyto funkce mají funkční argumenty, které odpovídají argumentům instrukce, a relokace těchto argumentů uvnitř funkcí se používají jako metadata při dynamickém překladu bloku virtuálních instrukcí (mezi dvěma sousedními skoky) na posloupnost fyzických instrukcí, které vzniknou spojením výkonných částí interpretačních funkcí po provedení příslušných relokací argumentů na virtuální paměťová místa, resp. paměťové obrazy virtuálních registrů.

Po sestavení každého takového bloku se ještě provádějí některé globální optimalizace jako odstranění výpočtů, jejichž výsledky nejsou později použity atd.

Zvláštní zřetel musí být při dynamickém překladu brán na samomodifikující se kód a další změny, které mohou způsobit, že přeložená posloupnost fyzických instrukcí již neodpovídá bloku instrukcí virtuálních. QEMU používá pro detekci samomodifikujícího se kódu ochranu paměti hostitelského operačního systému, kdy stránky paměti obsahující virtuální instrukce mapuje jen pro čtení a tím umožňuje efektivně detekovat pokusy o zápis.

5.2 Bochs

Simulátor Bochs se zaměřuje především na přesnou simulaci platformy IA-32 a AMD64, díky tomu, že je implementován v C++ a používá standardní interpretaci každé virtuální instrukce, je zároveň velmi dobře přenositelný. Tyto vlastnosti jej předurčují především jako nástroj pro vývoj a ladění jader operačních systémů, zavaděčů a dalšího nízkoúrovňového kódu.

5.3 Simics

Virtutech Simics je velmi univerzální a všestranný simulátor mnoha hardwarových platforem s širokou konfigurovatelností. Důraz je především kladen na maximální věrnost simulace a možnosti ladění, takže na rozdíl od ostatních simulátorů jsou brány v potaz i vnitřní stavy procesoru (cache, TLB apod.) a sběrnic.

5.4 PearPC

Primární motivací pro vznik tohoto simulátoru 32bitové varianty procesoru PowerPC byla možnost spouštění operačního systému Mac OS X na platformě IA-32. Tento cíl se podařilo splnit, byť možná za cenu toho, že simulátor implementuje jen skutečně nejnutnější minimum toho, co je pro splnění tohoto cíle potřeba, takže simulace procesoru, OpenFirmwaru a některých periférií je jen přibližná.

Procesor PowerPC G4 (750) je simulován dvěma způsoby: přenositelně pomocí sady metod v C++ (to je vhodné především pro ladění) a metodou dynamického překladu do instrukční sady IA-32 (tento překlad je ovšem naprogramován ručně v assembleru). Podporovány jsou též SIMD instrukce AltiVec (dynamický překlad na SSE), ovšem v kódu je několik drobných chyb, které způsobují chybné vykonávání některých instrukcí v konkrétních kontextech.

Z periférií je podporován standardní řadič přerušení a klávesnice VIA 6522, síťové karty RealTek 8139 a 3Com 90xC a generické rozhraní sběrnic PCI, MacIO, USB a IDE. Grafický výstup je urychlen volitelným použitím hypervolání společného s Mac-on-Linux (viz 6.6).

5.5 Rosetta

Aplikační emulátor Rosetta společnosti Transitive[1] je součástí operačního systému Mac OS X pro platformu IA-32. Používá velmi pokročilé metody dynamického překladu pro emulaci uživatelských instrukcí 32bitové varianty procesorů PowerPC (G3 a G4 včetně instrukční sady AltiVec) a umožňuje tak transparentní spouštění programů přeložených pro PowerPC i IA-32 v jednom operačním systému (systémová API se používají nativní IA-32, ale není možné například použít dynamické linkování programu pro PowerPC se sdílenou knihovnou pro IA-32 nebo obráceně).

Není bez zajímavosti, že podobnou metodu transparentní emulace použila společnost Apple také v nedávné historii při přechodu z platformy Motorola 68000 na PowerPC. Emulace probíhala na úrovni jádra a umožňovala, aby ještě dlouho po ukončení používání procesorů řady 68000 byla velká část jádra operačního systému Mac OS napsána v původním assembleru.

5.6 Virtual PC for Mac

Jedná se o simulátor platformy IA-32 pro operační systém Mac OS X, který simuluje stejné virtuální periférie jako virtualizér Virtual PC (viz 6.2), se kterým pravděpodobně sdílí také velkou část zdrojového kódu. Přesná metoda emulace instrukcí není ovšem veřejně dokumentována.

[1]Tato společnost nabízí také univerzální simulátor QuickTransit používající přenositelný dynamický překlad pro simulaci různých virtuálních instrukčních sad na různých fyzických strojích a současně překlad API různých operačních systémů (např. Solaris na GNU/Linux).

Kapitola 6

Přehled úplných virtualizérů

Obsahem této kapitoly je přehled úplných virtualizérů, v případě platformy IA-32 prakticky vyčerpávající.

6.1 VMware

Jedná se o historicky nejstarší úplný virtualizér pro platformu IA-32, v současné době nabízí také omezenou možnost virtualizace platformy AMD64 (vyžaduje pro to však podporu technologie Vanderpool nebo Pacifica). V hrubých obrysech odpovídá implementace mechanismu virtualizace metodu popsanou v sekci 3.2.3, je však použita celá řada velmi sofistikovaných „triků“, aby efektivita běhu virtuálního stroje byla co nejvyšší. Specifikace konkrétního typu operačního systému, který ve virtuálním stroji poběží, umožňuje, aby mohly být tyto sofistikované mechanismy optimalizované např. speciálně pro určitý převládající způsob správy virtuální paměti daným operačním systémem.

Nejstarší instance produktové řady VMware je VMware Workstation, virtualizér používající jako svůj hostitelský systém Windows (řada NT) nebo GNU/Linux. Volně šiřitelná varianta VMware Player má stejné možnosti virtualizace, ale neumožňuje vytvářet nové konfigurace virtuálních strojů, pouze používat konfigurace již předpřipravené.

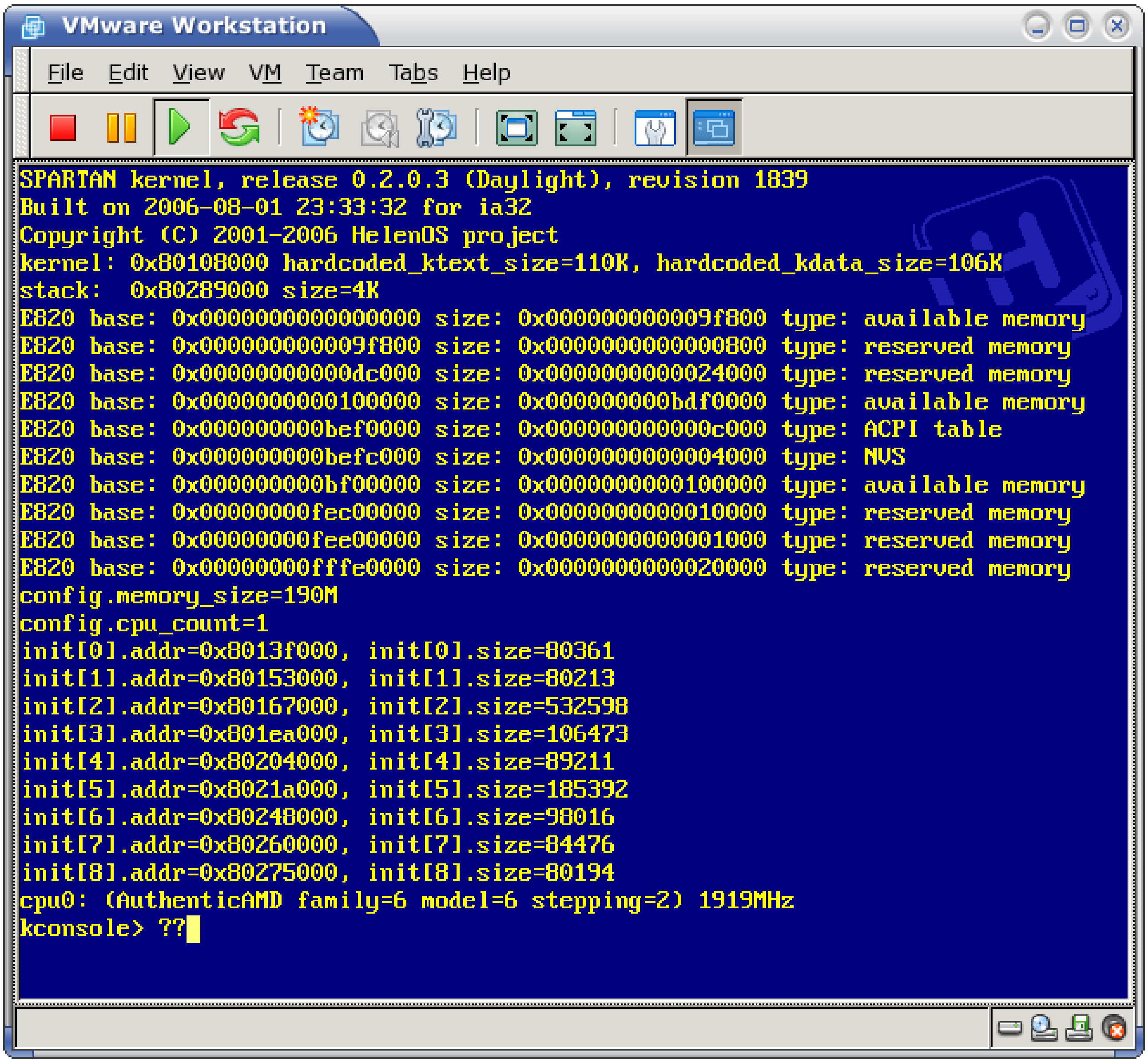
VMware Server (původně VMware GSX Server) je varianta, která má oddělenou část virtualizace od uživatelského rozhraní, zaměřuje se tedy na masivní provozování virtuálních serverů bez nutnosti interaktivní kontroly. Hostitelský systém je GNU/Linux nebo Windows 2003 Server. Od července 2006 je tento software zdarma ke stažení.

VMware ESX Server nepoužívá jako své fyzické rozhraní obecný hostitelský operační systém, ale upravené jádro SimOS (jádro Linuxu se používá pro inicializaci hardwaru).

6.1.1 Virtuální hardware

Ve virtuálním stroji je možno simulovat kromě základního hardwaru jako jsou sběrnice a klávesnice např. IDE a SCSI pevné disky. Jako fyzické úložiště pro tyto virtuální disky je možno použít disk fyzický nebo soubor na souborovém systému. V druhém případě je možno použít varianty jako neperzistentní disk (veškeré změny dat jsou po ukončení virtuálního stroje zrušeny a disk obsahuje data původní) nebo disk s možností návratu (veškeré změny je možno volitelně zrušit).

Další virtuální periférie jsou disketové mechaniky, sériové a paralelní porty, myš (PS/2) a zvukové karty (Sound Blaster 16). Virtuální síťová rozhraní (AMD PCnet) mohou být namapována na rozhraní fyzického počítače (bridging), nebo mohou být ve virtuální privátní IP podsíti a buď mít možnost komunikovat pouze s hostitelským operačním systémem, nebo pomocí překladu adres (NAT) také s okolní fyzickou sítí.



Obr. 6.1: Obrazovka VMware Workstation, hostitelský systém GNU/Linux, hostovaný systém HelenOS, platforma IA-32.

Fyzická SCSI a USB zařízení mohou být přímo namapována do virtuálního stroje, jejich použití je poté pro daný virtuální stroj exkluzivní. Virtuální USB rozhraní je verze 1.1.

Jako grafická karta je emulována generická SVGA s rozhraním VESA 3.0. Pro urychlení grafických operací je možno využít speciální hypervolání implementovaná jako zápisy do komunikační stránky, která funguje jako cyklická fronta grafických operací. Mezi nejdůležitější operace patří:

SVGA_CMD_RECT_FILL Vykreslení obdélníku danou konstantní barvou.
SVGA_CMD_RECT_COPY Zkopírování obdélníkové oblasti pixelů (bitblt).
SVGA_CMD_DEFINE_BITMAP Upload pixelů bitmapy (určené číselným identifikátorem).
SVGA_CMD_RECT_BITMAP_COPY Vykreslení bitmapy (určené číselným identifikátorem).
SVGA_CMD_DEFINE_CURSOR Změna „hardwarového“ kurzoru myši.

Pro řízení fronty (zasílání požadavku na provedení uložených příkazů), nastavování rozlišení, barevné hloubky a dalších parametrů, nebo naopak čtení stavových údajů (příznaku podporovaných operací a vlastností) slouží zápisy a čtení specifických I/O portů.

6.2 Virtual PC

Virtual PC, virtualizační produkt platformy IA-32 původně vyvíjený společností Connectix, byl v roce 2003 zakoupen společností Microsoft. Jako hostitelský operační systém je podporována v současné době pouze řada Windows počínaje verzí 2000, podporované hostované systémy jsou všechny operační systémy společnosti Microsoft pro IA-32 (včetně 16bitových verzí MS-DOSu), bez oficiální podpory, ale přesto spolehlivě v něm běží také další systémy (GNU/Linux, Solaris atd.). Od června 2006 je verze 2004 tohoto produktu ke stažení bez poplatku.

Procesor virtuálního stroje je svými vlastnostmi omezen na Pentium II, virtualizér používá dynamickou rekompilaci, kdy uživatelský kód chráněného módu a VM86 módu je až na kritické instrukce spouštěn beze změn, zatímco privilegovaný režim a kód reálného módu je upraven tak, aby nebyla narušena izolace virtuálního stroje. Tato metoda se tedy poněkud liší od způsobu nastíněného v sekci 3.2.3, v praxi je její efektivita srovnatelná. Disponuje-li procesor fyzického stroje rozšířením Vanderpool, dovede jej Virtual PC využít a tím proces dynamické rekompilace obejít.

Kromě základních periférií jsou ve virtuálním stroji k dispozici zvuková karta Sound Blaster 16, síťová karta DEC 21140 (Virtual PC však nedovede vytvořit privátní síť s NATem, pouze dovede použít některé z fyzických rozhraní) a grafická karta S3 Trio64V+ (která dovoluje přiměřenou míru akcelerace grafických operací i tam, kde virtualizovaný operační systém nemá k dispozici ovladač pro hypervisor grafické rozhraní, které je podobné jako u VMware).

Na rozdíl od produktů VMware nepodporuje USB a SCSI rozhraní a celkově jsou možnosti konfigurace virtuálního stroje menší.

6.3 Virtual Server

Produkt svým původem a vlastnostmi velmi podobný Virtual PC, ve verzi 2005 R2 byl v květnu 2006 uvolněn ke stažení bez poplatku. Jako hostitelský systém je podporován Windows XP a 2003 Server a to i na platformě AMD64 (virtuální stroj je však stále jen IA-32).

Mezi základní rozdíly patří to, že vstup a grafický výstup virtuálních serverů není směřován do okna desktopové aplikace, ale veškerá správa a „konzolová“ práce s virtuálními servery probíhá přes webové rozhraní (vyžadován je Internet Explorer 5.5 nebo vyšší) podobně jako u VMware Server.

6.4 Plex86

V současné době neudržovaný projekt snažící se vytvořit svobodný úplný virtualizér IA-32. Použitá metoda ošetření kritických instrukcí přesně odpovídá popisu v sekci 3.2.3. V současné době je ve virtuálním stroji podporován pouze GNU/Linux, což výrazně zužuje stavy, které je potřeba ošetřit.

6.5 Parallels Workstation

Parallels Workstation je vyvíjen společností Parallels, Inc., jehož vlastnosti (podpora fyzických strojů, vlastnosti virtuálního stroje a možnosti konfigurace virtuálních periférií, včetně hypervolání pro akceleraci grafických operací) jsou velmi podobné nebo až shodné s VMware Workstation.

Také dle zkoumání veřejně dostupných částí virtualizační metody (jaderné moduly pro Linux, které umožňují virtualizéru ovlivňovat privilegované stavy fyzického stroje a odchytávat výjimky způsobené kódem virtuálního stroje prováděným v uživatelském režimu procesoru) se zdá, že i implementační detaily se VMware velmi podobají (specifická dokumentace není veřejně k dispozici).

6.6 Mac-on-Linux

Úplná virtualizace pro 32bitové procesory PowerPC 603, 604, G3 a G4. Jako hostitelský operační systém je vyžadován výhradně GNU/Linux, mezi podporované hostované systémy ve virtuálním stroji patří GNU/Linux, Mac OS 7.5.2 až 9.2.2 a Mac OS X 10.1 až 10.3.3. Obsahuje jen částečnou implementaci OpenFirmware.

Virtuální periférie obsahují kromě standardních systémových součástí (řadič přerušení, klávesnice, sběrnice) také grafický adaptér (pomocí ovladače používajícího hypervolání je možné některé grafické operace akcelerovat), síťový adaptér, zvukovou kartu a IDE diskové rozhraní. SCSI a USB zařízení fyzického stroje je možné použít přímo ve virtuálním stroji.

Platforma PowerPC splňuje Popek-Goldbergovy dostačující podmínky úplné virtualizace, takže implementace Mac-on-Linux je velmi přímočará – virtuální stroj je realizován jako běžný uživatelský proces, zatímco jaderný modul se stará o ošetření výjimek, způsobených pokusem o provádění privilegovaných instrukcí, emulací těchto instrukcí a případně změnou příslušných vnitřních stavů virtuálního stroje. Poměrně velká část kódu modulu slouží k efektivnímu využití mechanismu stránkování fyzického stroje pro virtualizaci stránkování virtuálního stroje.

Kapitola 7

Přehled paravirtualizérů

Paravirtualizér Xen, jehož detailnější popis zabírá převážnou část této kapitoly, slouží jako modelový příklad implementace paravirtualizace.

7.1 Xen

Paravirtualizér Xen vyvíjený týmem z Univerzity v Cambridge je aktuálně ve verzi 3.0.2. Mezi podporované platformy fyzického a současně virtuálního rozhraní patří IA-32 a AMD64 (IA-64 je v raném stádiu vývoje). Součástí zdrojového kódu je také podpora pro technologie Vanderpool a Pacifica, takže na fyzickém procesoru, který je některou z těchto technologií vybaven, je možné provozovat úplnou virtualizaci neupraveného operačního systému.

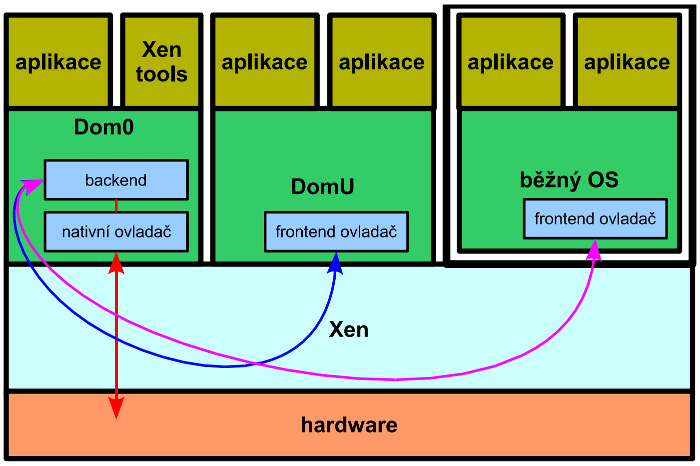
Jednotlivé virtuální stroje se v terminologii Xenu nazývají *domény* a existuje značný principiální rozdíl mezi takzvanou iniciální doménou (*Dom0*) a dalšími uživatelskými doménami (*DomU*). Samotný Xen je navržen jako mikrojádro, takže jeho fyzické rozhraní komunikuje přímo s hardwarem. Přímou podporu má Xen ovšem pouze pro hardware nezbytně nutný k realizaci paravirtualizace (procesor, paměť, textová konzole, sériová konzole), o veškerý další hardware se stará operační systém Dom0, který k němu může získat fyzický přístup.

Další rolí Dom0 je vytvořit rozhraní (backendy), kterými mohou uživatelské domény (DomU) přistupovat k prostředkům fyzického stroje (pomocí frontend ovladačů). Xen tedy realizuje pouze izolaci jednotlivých domén, roli privilegovaného správce prostředků přebírá operační systém běžící jako Dom0.

7.1.1 Virtualizace paměti

Překlad paměťových adres každé domény probíhá standardním mechanismem stránkování – virtuální adresa (přesněji řečeno lineární adresa) se pomocí stránkovacích tabulek převádí na adresu fyzickou. Paravirtualizace paměti probíhá tak, že doména nemá přiřazenu souvislou oblast fyzických rámců, ale různé rámce fyzické paměti. Abstrakce souvislé fyzické paměti je vytvořena překladovou tabulkou P2M, která převádí abstraktní rámce na skutečné (strojové) rámce fyzické paměti. Pokud potřebuje doména namapovat virtuální stránku v na abstraktní fyzický rámec p, musí jej ve skutečnosti namapovat na fyzický rámec $m = \mathrm{P2M}(p)$.

Aktualizace stránkovacích tabulek probíhá dvěma možnými mechanismy: explicitním použitím hypervolání MMU_UPDATE, které modifikuje příslušné záznamy stránkovacích tabulek, nebo přímým zápisem do stránkovacích tabulek nižší než nejvyšší úrovně (stránky těchto tabulek jsou chráněny před zápisem, takže pokus o jejich modifikaci vyvolá výjimku, kterou odchytí hypervisor, příslušnou tabulku vyřadí z mechanismu stránkování a umožní do ní následný zápis – k ověření správnosti hod-



Obr. 7.1: Tři druhy operačních systémů v prostředí Xen: zleva Dom0 fungující jako správce zdrojů, přeportovaný DomU používající speciální ovladače a nemodifikovaný operační systém běžící v prostředí s hardwarovou podporou virtualizace používající také speciální ovladače.

not a opětovném připojení stránky do mechanismu stránkování dojde při nejbližší výjimce na jiné stránce).

7.1.2 Komunikace

Komunikace mezi hypervisorem a virtuálním strojem probíhá několika možnými způsoby.

Výjimky jsou virtuálnímu stroji doručovány analogicky jako na stroji fyzickém, pouze mechanismus nastavení obslužných rutin není přímý (změnou hodnoty v IDT), ale pomocí hypervolání SET_TRAP_TABLE.

Kanály událostí

Pro doručování informací o hardwarových přerušeních a dalších asynchronních událostech, stejně jako pro zpětnou notifikaci hypervisoru se používá mechanismus kanálů událostí. Pro doručování událostí musí doména registrovat callback funkci, která funguje podobně jako obsluha přerušení na fyzickém stroji. Událost je chápána jako změna stavu z hodnoty 0 na hodnotu 1 (level triggered).

Rozhraní pro manipulaci s kanály událostí umožňuje (v závislosti na oprávnění aktuální domény):

- Alokovat nový lokální port.
- Připojit lokální port kanálem na jiný port jiné domény.
- Připojit lokální port kanálem na virtuální přerušení.
- Připojit lokální port kanálem na fyzické přerušení.
- Připojit lokální port kanálem na jiný virtuální procesor pro doručování IPI zpráv.
- Omezení doručování zpráv z kanálu jen na konkrétní virtuální procesor.
- Odeslání události kanálem.

Kromě toho je možné maskovat přijímání událostí konkretním kanálem analogicky jako se zakazují na fyzickém stroji přerušení.

Xenstore

Xenstore představuje jakýsi jednoduchý sdílený souborový systém v paměti, který slouží pro předávání stavových údajů a informací mezi jednotlivými doménami.

7.1.3 Proces bootování

Mikrojádro Xen je přeloženo jako komprimovaný ELF, takže k jeho nabootování se obvykle používá boot loader GRUB nebo jiný zavaděč odpovídající multiboot specifikaci. Jako povinný bootovací modul je uvedeno jádro operačního systému Dom0. Volitelně další moduly představují moduly tohoto jádra (nebo v případě Linuxu initrd).

Fyzicky je obraz jádra načten na adresu 1 MB a spuštěn v privilegovaném režimu procesoru (přesněji ring 0) s povoleným stránkováním. Prvních 64 MB fyzické paměti je vyhrazeno pro potřeby Xenu (32 MB pro struktury sdílené s doménami, 32 MB pro pracovní haldu, zásobníky a kód). Fyzická paměť je po bootu mapována jednak identicky 1:1, dále je prvních 64 MB fyzické paměti namapováno od virtuální adresy 0xFC000000 (resp. 0xF5800000 v případě použití PAE[1]). Samotný kód je relokován na adresu 0xFF000000.

Po inicializaci správy paměti dojde na inicializaci hardwaru, což představuje:

- Nalezení výstupní konzole. Výstupní konzole může být buď standardní EGA/VGA textový framebuffer a/nebo sériový port.
- Rozpoznání aplikačních procesorů (na základě údajů z ACPI tabulek) ve víceprocesorových systémech a jejich nastartování.

7.1.4 Načtení jádra Dom0

Podle údajů z ELF hlavičky jádra Dom0 je vytvořeno identické mapování paměti začínající na virtuální adrese nejnižší bázové adresy všech ELF sekcí. Tato adresa musí být větší než 1 GB a je přitom zaokrouhlena na velikost 4 MB dolů.

Za touto oblastí jsou dále identicky namapovány další oblasti:

- Volitelně další zaváděné moduly.
- Tabulka P2M.
- Struktura `start_info_t`.
- Zaváděcí stránkovací tabulky (popisující aktuální mapování).
- Zaváděcí zásobník.
- Volná paměť (alespoň 512 KB).

Každá tato oblast je zarovnána na 4 KB a volná paměť na konci je zarovnána na 4 MB. Dále je do virtuálního adresového prostoru nové domény namapována část paměti začínající na adrese 0xFC000000 (resp. 0xF5800000 v případě použití PAE), která obsahuje tabulku M2P (pro převod skutečných strojových rámců na rámce abstraktní). Tato tabulka je pochopitelně sdílená mezi všemi doménami (převod není prostý).

Zaváděné jádro musí obsahovat speciální sekci `__xen_guest`, která obsahuje textový řetězec s parametry pro zavádění. Jedná se o seznam záznamů ve tvaru `proměnná=hodnota` oddělených čárkou. Některé významné hodnoty:

XEN_VER Verze rozhraní paravirtualizéru, které jádro předpokládá. Pokud není rozhraní kompatibilní s aktuálně běžící implementací Xenu, není jádro spuštěno.

[1] Physical Address Extension – rozšíření IA-32 o podporu 36bitových fyzických adres pomocí 4úrovňového stránkování.

LOADER Typ zavaděče, který se má použít. Slouží pro speciální ošetření nestandardních jader.
HYPERCALL_PAGE Číslo abstraktního fyzického rámce, do kterého bude vsunut kód pro hypervolání.

7.1.5 Spuštění Dom0

Po načtení jádra a vytvoření mapování je vytvořena příslušná sada deskriptorů pro běh v režimu procesoru ring 1 (limit segmentů je nastaven tak, aby nebylo možné přistupovat do nesdílených stránek Xenu), do stránky určené hodnotou HYPERCALL_PAGE (případně symbolem `hypercall_page`, obsahuje-li ELF tabulku globálních symbolů) je nahrán kód pro hypervolání a řízení je předáno vstupnímu bodu jádra (registr ESI obsahuje virtuální adresu struktury `start_info_t`, registr ESP je nastaven na vrchol zaváděcího zásobníku).

Struktura `start_info_t` obsahuje informace potřebné pro základní běh domény, podstatné jsou tyto položky:

nr_pages Celkový počet abstraktních fyzických stránek, které má doména k dispozici.
shared_info Fyzická adresa struktury popisující konfiguraci virtuálního stroje (počet virtuálních procesorů, kanály zpráv, zdroj lokálního a globálního času atd.).
store_mfn Číslo fyzické stránky pro sdílené úložiště.
console_mfn Číslo fyzické stránky vstupně/výstupní konzole.
pt_base Virtuální adresa stránkovací tabulky nejvyšší úrovně (adresáře).
nr_pt_frames Počet stránek (za stránkou obsahující stránkovací tabulku nejvyšší úrovně), které jsou použity stránkovacími tabulkami nižší úrovně.

Spuštěné jádro operačního systému obvykle provede překopírování těchto informací do svých struktur a inicializuje mapování abstraktních rámců na vhodné virtuální adresy.

7.1.6 Hypervolání

Hypervolání služeb Xen se provádí zavoláním příslušné funkce uložené ve stránce HYPERCALL_PAGE. Každé funkci je vyhrazeno 32 bytů a pomocí instrukce INT, SYSENTER nebo skoku na volací bránu předávají řízení hypervisoru a předávají mu argumenty uložené v registrech.

Základní hypervolání jsou tyto:

MMU_UPDATE Aktualizace paměťového mapování, modifikuje konkrétní položku stránkovací tabulky (určené fyzickou adresou) na příslušnou hodnotu (obsahující opět fyzickou adresu). Xen ověřuje, zda je použité mapování korektní a zda fyzická stránka skutečně patří příslušnému virtuálnímu stroji.
SET_TRAP_TABLE Registruje pro virtuální stroj obslužné rutiny virtualizovaných výjimek a přerušení. Čísla výjimek a přerušení odpovídají hodnotám jejich fyzických ekvivalentů.
EVENT_CHANNEL_OP Posílá Xenu zprávu na kanálu událostí.
MMUEXT_OP Umožňuje změnit virtuální GDT, LDT, případně nastavit fyzickou adresu nové stránkovací tabulky nejvyšší úrovně (virtuální CR3).

7.1.7 Víceprocesorové systémy

Plánovač Xenu podporuje víceprocesorové systémy včetně technologie hyper-threadingu. Na každý fyzický procesor je namapován jeden procesor virtuální, přičemž doméně může být přidělen pevný nebo pohyblivý počet těchto virtuálních procesorů. IPI komunikace mezi procesory je realizována ve virtuálním stroji pomocí kanálů událostí.

7.1.8 Virtualizace zdrojů

Variant realizace správy zdrojů je několik a liší se také podle toho, zda ke zdrojům přistupuje doména Dom0 nebo DomU. Ve všech případech je možné Xen nastavit tak, aby doménám umožnil bezpečný přístup přímo k fyzickým prostředkům (v případě sběrnice PCI poskytuje například virtuální konfigurační prostor obsahující povolenou podmnožinu fyzických zařízení).

Další varianta představuje správu zdrojů pomocí Dom0, jak ukazuje obrázek 7.1. Systém běžící jako Dom0 má fyzický přístup k hardwaru a poskytuje k němu pomocí backendů rozhraní pro frontend ovladače ostatních domén. Předávání dat probíhá sdílenou pamětí.

Xen také definuje několik vlastních rozhraní, např. pro virtuální síťová a bloková zařízení.

7.1.9 Fungování s dvěma režimy procesoru

Platforma AMD64 v 64bitovém režimu nepodporuje 4 režimy procesoru jako v 32bitových režimech kompatibilních s IA-32. Jádro virtualizovaného operačního systému stejně jako jeho uživatelské procesy proto musí běžet v neprivilegovaném režimu a jsou vzájemně rozlišeny různými stránkovacími tabulkami. Stránkovací tabulky uživatelských procesů obsahují pouze uživatelské stránky, zatímco stránkovací tabulky jádra obsahují uživatelské stránky i privilegované stránky jádra.

7.2 Denali

Paravirtualizační jádro Denali vychází z poněkud jiných předpokladů než ostatní paravirtualizační přístupy. Shodné rysy jsou v tom, že instrukce neprivilegovaného režimu IA-32 jsou prováděny přímo, zatímco instrukce ovlivňující stav virtuálního stroje musí být ve virtualizovaném jádře operačního systému nahrazeny voláními hypervisoru.

Virtualizují se také fyzická přerušení, pro komunikaci mezi operačním systémem a hypervisorem se používá sdílená paměťová oblast. Virtualizace paměti je ovšem výrazně zjednodušená, nepoužívá se segmentace a celé jádro operačního systému pracuje v jediném plochém adresním prostoru (stránkování se používá čistě na úrovni hypervisoru pro ochranu jeho kódu, dat a paměti ostatních virtuálních strojů). Důsledkem toho musí být jádro operačního systému staticky slinkováno s aplikačním programem, který se ve virtuálním stroji spouští, což celý mechanismus přibližuje koncepci exokernelu.

Autoři uvádějí (viz [11]), že tento přístup je v pořádku, pokud přesuneme požadavek na existenci více prstenců ochrany z operačního systému do hypervisoru.

Na rozdíl od Xenu pracuje jádro Denali také jako správce zdrojů, obsahuje tedy ovladače pro konkrétní periferie a vůči operačnímu systému poskytuje jen abstraktní rozhraní shodná vždy pro danou třídu zařízení. Jeden z virtuálních strojů běžících pod Denali má práva pro běhovou konfiguraci hypervisoru, umožňuje spouštět nové virtuální stroje atd.

Značnou nevýhodou tohoto projektu v porovnání s ostatními paravirtualizačními prostředky je jeho uzavřenost a malá uživatelská základna.

7.3 UML

User-Mode Linux nevznikl původně jako implementace paravirtualizace pro Linux, ale jako prostředek pro snadné ladění Linuxového jádra během jeho vývoje. Původní implementace umožňovala spustit upravené jádro Linuxu jako běžný uživatelský proces běžící pod jádrem hostitelským. Procesy běžící v rámci hostovaného jádra (z pohledu hostujícího jádra se jednalo o vlákna) s ním poté sdílely adresní prostor a běžný mechanismus ochrany paměti musel být zastoupen složitými testy a krokováním procesu v některých případech.

Nová implementace od jádra verze 2.6.0 vytváří pro hostované jádro separátní privilegovaný adresní prostor.

7.4 TRANGO

Paravirtualizér pro platformy ARM, MIPS, PowerPC a SuperH, zaměřený především na embedded zařízení. Je implementován jako mikrokernel, který se stará o přidělování přístupových práv k fyzickým zdrojům, samotné řízení hardwaru provádějí poté jádra operačních systémů v jednotlivých virtuálních strojích standardními ovladači. Hypervolání a komunikace mezi jednotlivými virtuálními stroji je realizována pomocí předávání zpráv.

Pro TRANGO je portován Linux a eCos.

Kapitola 8

Přehled implementací partitioningu

Stejně jako několik předchozích kapitol je i tato věnována z velké části popisu jedné konkrétní implementace partitioningu, za nímž následuje stručnější výčet dalších produktů

8.1 Linux VServer

Linux VServer je rozšíření jádra Linuxu, které do něj přidává podporu partitioningu. Jeho použití na GNU/Linuxu je plně kompatibilní s použitím jádra Linuxu bez tohoto rozšíření, prostředky jádra jsou jen rozděleny do kontextů. Každý kontext se uživatelským procesům jeví jako samostatný virtuální stroj. Pro usnadnění administrace a sjednocení celého modelu jsou po nabootování všechny objekty v implicitním kontextu 0 (*host context*).

Oddělení kontextů je realizováno pomocí několika prostředků:

- **Izolace procesů**
  Procesy z různých kontextů se nevidí (ve výpisu procesů, v `/proc`, nelze jim posílat signály apod.). Výjimkou je proces `init`, který může být volitelně viditelný ve všech kontextech (kvůli kompatibilitě s některými utilitami), ovšem nelze mu posílat signály. Je také ošetřeno, že při vzniku nového kontextu není možné proces, který v něm vzniká, krokovat.
- **Izolace dalších zdrojů**
  Mezi další zdroje systému, které je potřeba mezi jednotlivými kontexty izolovat, patří sdílená paměť a další prostředky SYSV IPC, virtuální terminály, sockety atd.
- **Ireversibilní chroot**
  Procesy běží nad sdíleným souborovým systémem, každému kontextu lze přitom přiřadit jiný adresář jako kořenový adresář. K tomu se používá standardní mechanismus chroot (change root), společný nadadresář všech kontextů (typicky `/vservers`) je chráněn tzv. chroot bariérou, která upravuje sémantiku systémového volání `chroot`, aby nebylo možné nový kořenový adresář opustit.
- **Omezení IP adres**
  K fyzickým síťovým rozhraním jsou vytvořeny aliasy s dalšími IP adresami a jednotlivé kontexty jsou omezeny tak, že lze použít funkci `bind()` jen na adresy konkrétních aliasů. Tím lze efektivně přidělit různým kontextům různé IP adresy. Pouze localhost (tedy rozsah 127.0.0.1/8) musí být z principu sdílen mezi všemi kontexty (to lze ovšem obvykle vyřešit administrativně na úrovni konfigurace jednotlivých virtuálních serverů).
  Linux VServer v současné době podporuje jen IPv4.
- **Capability ceiling**
  Maska, která umožňuje omezit jaderná oprávnění (capabilities) uživatele root (a dalších uživatelů) kontextu. Mezi nejpodstatnější omezení patří zákaz rebootu, vytváření device nodů,

přístupu k blokovým zařízením, připojování souborových systémů, změny síťových parametrů a obecných parametrů jádra, které ovlivňují také ostatní kontexty (routovací tabulky, netfilter atd.).

- **Globální limity**
  Celkové omezení prostředků, které může virtuální server využít. Paměť, počet procesů, počet otevřených souborů atd (analogicky jako nastavení ulimitu pro uživatele).

Kromě běžných kontextů existuje v systému ještě speciální kontext 1 (*spectator context*), ve kterém se neuplatňují žádné kontextové testy a tudíž má přístup ke všem entitám systému. Spustit proces v tomto speciálním kontextu ovšem může pouze uživatel root z kontextu 0.

Linux VServer umožňuje provozovat různé distribuce uživatelského systému. Fyzický stroj, jádro a proces `init` je pro všechny kontexty společný, veškeré ostatní součásti (daemony, způsob spravování služeb pomocí init skriptů, virtuální terminály, sdílené knihovny atd.) jsou již zcela nezávislé.

Režie partitioningu je velmi malá (zhruba 2 % výkonu), je to vykoupeno některými omezeními. Standardní jaderná oprávnění nejsou v některých případech dostatečně jemná, bylo vhodné je rozšířit o další speciální případy, například možnost povolení jen některých specifických druhů raw síťových paketů (pro DHCP). Problém se sdílením společné síťové vrstvy mezi všemi kontexty způsobuje, že existuje jen jediný společný síťový localhost a není tedy možné, aby na shodném portu rozhraní 127.0.0.0/24 běžely služby z více kontextů. Toto omezení lze ovšem administrativně vyřešit tak, že služby budou používat vždy vnější síťové rozhraní.

8.1.1 Implementace

Správa kontextů se provádí sadou uživatelských utilit, které s jádrem komunikují pomocí speciálního systémového volání.

Izolace jednotlivých kontextů je dosaženo poměrně nevelkými změnami zdrojového kódu běžného jádra Linuxu, nejvíce změněného kódu je v jednotlivých souborových systémech, do kterých se přidává podpora atributu XID (viz dále). Z jaderných struktur jsou identifikátorem kontextu rozšířeny struktury pro aktuálně naplánovaný proces, ostatní procesy, sockety, IPC primitiva, uživatele a síťové prostředky. Testy na viditelnost a možnost interakce entit jsou omezeny jen na několik málo míst jádra.

Pro uchovávání globálních parametrů kontextu slouží struktura `struct vx_info`, která mimo jiné umožňuje reprezentovat stromovou hierarchii kontextů.

8.1.2 Sdílení částí souborového systému

Aby bylo možné použít část souborového systému pro více kontextů současně bez omezení míry izolace, nabízí Linux VServer dvě ortogonální metody.

Unifikace

Pro sdílení shodného obsahu souborů (obvykle binárních souborů v `/usr/bin`, `/usr/lib` apod.) lze použít metodu unifikace. Speciální utilita projde soubory se stejným názvem v jednotlivých kontextech a tam, kde zjistí také shodu obsahu souborů, je změní na hardlinky (na společný i-node) a nastaví se jim speciální příznak *immutable-linkage-invert*. Ten způsobí, že v případě pokusu o změnu (nebo unlink) dojde k vytvoření kopie i-nodu.

Obsah unifikovaných souborů je tedy v souborovém systému uložen pouze jednou a v případě, že v některém kontextu dojde k pokusu o změnu v některém kontextu, bude změna provedena v privátní kopii obsahu. Příznak *immutable-linkage-invert* je zatím implementován pro souborové systémy ext2, ext3 a reiserfs.

XID

Pro sdílení adresářů mezi jednotlivými kontexty tak, že každý kontext vidí pouze adresářové položky souborů, které vlastní, lze souboru přiřadit atribut XID. Tato metoda umožňuje také nad sdíleným souborovým systémem vynucovat kvóty pro jednotlivé kontexty.

Na souborových systémech ext2 a ext3 je atribut XID implementován jako nová položka i-nodu, což umožňuje zachovat plný rozsah standardních UID a GID atributů souborů. U ostatních souborových systémů lze vyhradit některé horní bity UID/GID pro kódování 16bitového XID (rozdělení 32:16, 16:32 nebo 24:24).

8.1.3 Plánování

Pro tvrdé omezení výpočetního výkonu, který může kontext spotřebovat, disponuje Linux VServer rozšířením plánovače na principu žetonů. Každý kontext má přihrádku pro určitý maximální počet žetonů, do které jsou v konstantních časových intervalech přidávány žetony, a naopak za každé časové kvantum, po které je nějaký proces daného kontextu naplánován, je jeden žeton odebrán.

Klesne-li počet žetonů některého kontextu na nulu, přestanou být jeho procesy plánovány, dokud počet žetonů nestoupne na nenulovou minimální hodnotu. Volitelně je možné podle aktuálního počtu žetonů ovlivňovat také prioritu procesů kontextu.

8.1.4 Základní práce s kontexty

Pro základní práci s kontexty lze použít příkaz `chcontext`, jehož základní syntaxe je následující:

```
chcontext [--cap capability] [--cap !capability]
          [--xid context] [--flag flag] [--secure] prog
```

Utilita spustí program zadaný cestou **`prog`** v novém kontextu. Uživatel root z kontextu 0 může pomocí parametru `–xid` specifikovat číslo kontextu explicitně. Pomocí přepínačů `–cap` lze nastavit capability ceiling, přepínač `–secure` odstraní z capability ceiling všechna potenciálně nebezpečná oprávnění.

Přepínač `–flags` (lze použít vícekrát) umožňuje nastavit příznaky nového kontextu:

- `fakeinit`
  Jako proces s PID 1 bude v kontextu vidět `/sbin/init`.
- `lock`
  Z kontextu nebude možné vytvořit další kontext.
- `private`
  Do kontextu nebude možné „vložit“ pomocí dalšího volání `chcontext` s přepínačem `–xid` nový proces. Nový proces může vzniknout jen systémovým voláním `fork` uvnitř kontextu.
- `ulimit`
  Aktuální nastavení ulimitu se budou aplikovat jako omezení na celý kontext.
- `virt_uptime`
  Hodnota uptime bude v rámci kontextu počítána od jeho vytvoření.

8.2 OpenVZ

OpenVZ představuje alternativní implementaci kontextů[1] pro jádro Linuxu, poskytuje však některé pokročilejší vlastnosti než Linux VServer. Mezi méně podstatné rozdíly patří jiný způsob imple-

[1] V terminologii OpenVZ se nazývají *virtuální prostředí* (Virtual Environments).

mentace limitů jednotlivých kontextů (místo aplikace ulimitů na celý kontext se používá separátní mechanismus) a jiná implementace limitujícího plánovače.

Změny oproti standardnímu jádru Linuxu jsou ovšem mnohem větší, kromě izolace procesů a dalších zdrojů v jednotlivých kontextech dochází ke skutečné „vnitřní virtualizaci“ jaderných zdrojů – stránkovací mechanismus je rozšířen o barvení stránek podle kontextů, identifikace procesů (PIDy) jsou unikátní jen v rámci kontextu, obslužné rutiny přerušení musí explicitně nastavovat globální kontext, atd.

Všechny tyto změny slouží k tomu, aby bylo možné nejen odizolovat jednotlivé kontexty mezi sebou, ale také zapouzdřit každý kontext podobně jako by se jednalo o celý virtuální stroj v případě paravirtualizace nebo úplné virtualizace.

Negativním důsledkem tak radikálních zásahů do struktury standardního jádra Linuxu je to, že se autorům nedaří udržovat krok s rychlým vývojem jádra a stabilní verze modifikovaných jader vycházejí z poměrně neaktuálních verzí (což může být problém například z hlediska bezpečnostních chyb a zranitelností).

8.2.1 Checkpointing a živá migrace

Ojedinělou vlastností OpenVZ je možnost uložení kompletního stavu kontextu (stav všech procesů, otevřených souborů, sít'ových spojení atd.) do souboru a jeho případné přesunutí na jiný fyzický stroj.

Při migraci dochází jen ke krátkému výpadku služby.

8.2.2 Virtuozzo

Komerční řešení partitioningu, které jako svůj základ používá OpenVZ. Kromě podpory GNU/Linuxu je Virtuozzo také ojedinělá implementace partitioningu pro serverové systémy Microsoft Windows.

8.3 FreeBSD Jails

Jails je subsystém jádra FreeBSD, který se svými vlastnostmi velmi podobá Linux VServeru. Základní součásti představuje mechanismus *securelevel* a izolace procesů podobná jako v případě standardního unixového volání chroot. Na rozdíl od VServeru se jedná o standardní součást jádra (od verze 4.0-RELEASE), neposkytuje ovšem některé pokročilé možnosti (sdílení společného souborového systému, tvrdé limity využívaných prostředků atd.) a stejnou flexibilitu.

Bezpečnostní mechanismus securelevel je implementován jako číselná proměnná náležející každému procesu, která se vzrůstající hodnotou omezuje více oprávnění superuživatele. Tuto hodnotu lze jen zvyšovat, pouze proces `init` ji může opět snížit. Každá partition má navíc vlastní hodnotu securelevel a při testu oprávnění se vždy uvažuje ta vyšší z hodnot náležejících procesu a partition.

Na rozdíl od VServeru po nabootování jádra nemají procesy žádnou speciální vlastnost. Proces může pomocí systémového volání `jail`(2) vstoupit do izolované partition, kterou už nemohou opustit. Stejnou partition potom sdílejí také všechny synovské procesy.

Partition omezuje procesy v těchto ohledech:

- Přístup k souborovému systému je omezen na podstrom podobně jako u mechanismu chroot.
- Je možno použít funkci `bind()` jen na jednu konkrétní IP adresu, argument *any address* je automaticky převeden na tuto adresu.
- Práva uživatelů jsou omezena, včetně práv uživatele root. V rámci partition není možné:
  - Modifikovat jádro systému, jeho parametry nebo do něj zavádět moduly.

- Modifikovat konfiguraci síťových rozhraní a směrovací tabulky
- Připojovat a odpojovat souborové systémy.
- Vytvářet uzly zařízení.
- Využívat některé druhy síťových socketů (raw, divert apod.).
- Modifikovat souborové příznaky *securelevel*.

- Veškeré interakce s ostatními procesy je omezena na společnou partition.

8.3.1 Implementace

Velmi analogicky s Linux VServer je jaderná struktura každého procesu `struct proc` rozšířena o ukazatel na strukturu typu `struct prison` popisující danou partition. Při vzniku synovského procesu je ukazatel na tuto strukturu děděn.

Nastavení ukazatele na `struct prison` a naplnění struktury hodnotami je možné výhradně pomocí systémového volání `jail` a to jen v případě, že proces zatím na žádnou strukturu neukazuje. S výjimkou dědění partition se vždy vytváří nová, není tedy možné „vsunout“ proces do již existující partition.

Funkce jádra FreeBSD `p_trespass(p1, p2)` slouží na testování, zda proces `p1` může ovlivnit proces `p2`; v případě, že je jeden z procesů v partition, testuje se také, zda je druhý proces ve stejné partition.[2] Viditelnost procesů je ovlivněna různými pohledy na souborové systémy `procfc` a `sysctl` v každé partition.

Omezení dostupných IP adres je řešena existujícími mechanismy TCP stacku FreeBSD, rozhraní pro nastavování a čtení konfigurace síťových rozhraní byla rozšířena o specifické pohledy podle konkrétní partition a byla zablokována možnost změny konfigurace v rámci partition. Podobně musel být upraven kód pro přístup k některých systémovým prostředkům, především virtuálním terminálům, systémovému volání `mknod` a dalším zhruba 260 jednotlivým případům, kdy získával uživatel root zvýšená privilegia oproti běžnému uživateli.

8.3.2 Uživatelská správa

Stejně jako jaderná implementace mechanismu Jails je i jeho uživatelské použití velmi přímočaré. Pro vytvoření a spuštění nové partition se potřeba do zvoleného adresáře (např. `/jail/part1`) instalovat běžné knihovny a uživatelské programy systému FreeBSD včetně jejich konfigurace.

Následně se do tohoto podstromu připojí souborový systém `procfs`, k jednomu fyzickému síťovému rozhraní se vytvoří alias s novou IP adresou a spustí se standardní startovací skript `/etc/rc` v prostředí nové partition. Vše jednoduše demonstrují tyto tři příkazy:

```
# ifconfig ed0 inet add 192.168.1.2 netmask 255.255.255.255
# mount -t procfs proc /jail/part1/proc
# jail /jail/part1 part1 192.168.1.2 /bin/sh /etc/rc
```

8.4 Solaris Containers

Technologie partitioningu, která je k dispozici od Solarisu verze 10, používá několik technik s podobnými rysy jako v případě Linux VServeru nebo FreeBSD Jails.

[2] Z konstrukce tohoto testu plyne, že procesy, které nejsou součástí žádné partition, nemají vůči procesům v partition omezenou viditelnost. Analogickou roli má u Linux VServeru *spectator context*.

Jednotlivé kontexty se označují jako *zóny*, přičemž procesy spuštěné v systému po bootu jsou součástí globální zóny.

Kapitola 9

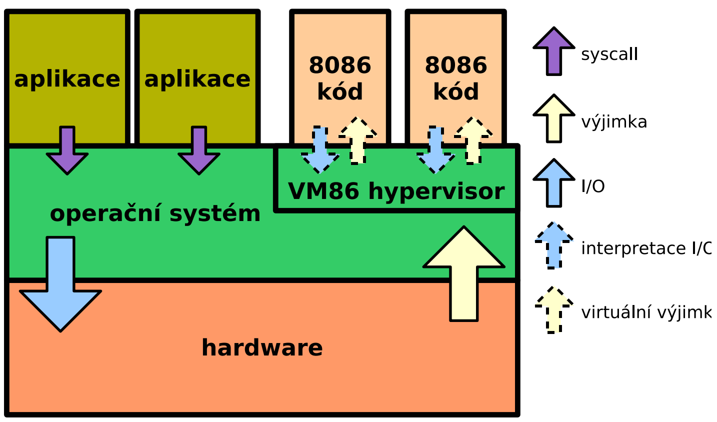
Hardwarová podpora virtualizace

Specifické technologie pro hardwarovou podporu virtualizace slouží k dvěma rozdílným účelům: na platformách, které původně nesplňují Popek-Goldbergovy dostačující podmínky úplné virtualizace (nebo dokonce ani nutné podmínky úplné virtualizace) tento nedostatek řeší (to je motivace pro vytvoření technologií Vanderpool a Pacifica). Na platformách, kde je úplná virtualizace možná, slouží tato rozšíření k její snadnější implementaci nebo efektivnější realizaci, například tím, že přidávají další úroveň ochrany paměti a překladu adres apod. (to je případ technologie Power Hypervisor).

9.1 VM86

V mikroprocesoru Intel 80386 a dalších následujících procesorech architektury IA-32 je implementován mechanismus virtualizace *Virtual 8086*, který umožňuje virtualizovat běh aplikací a operačních systémů pracujících v takzvaném reálném módu procesoru (zjednodušeně řečeno využívajících instrukční sady a vlastností procesoru 8086, resp. instrukční sady reálného módu následujících procesorů).

Tento mechanismus se používá především k simulaci běhového prostředí starších aplikací, ale také například k bezpečnému provádění rutin VGA BIOSu v rámci operačního systému běžícího v chráněném módu. Bez jeho pomoci není dostatečná izolace kódu běžícího v reálném módu procesoru od zbytku operačního systému prakticky možná. Naopak způsoby pro jeho využití pro virtualizaci operačních systémů běžících v chráněném módu jsou prakticky nulové, takže hypervisor VM86 běží obvykle jen jako součást operačního systému.



Obr. 9.1: Virtual 8086.

Kód napsaný pro reálný mód by bylo možné spouštět přímo v chráněném módu (v jeho 16bitové variantě kompatibilní s 80286), pokud by dodržel několik podmínek:

- Nepoužíval by segmentovou aritmetiku a nepředpokládal by, že se segmenty paměti překrývají.
- Nepoužíval by privilegované instrukce a nepřistupoval přímo k periferiím.
- Nezapisoval by do kódových segmentů (což vylučuje samomodifikující se kód).
- Nepokoušel by se spouštět kód z datových segmentů.

Tyto podmínky typicky kód psaný pro reálný mód nesplňuje, proto bylo potřeba jeho bezpečné provádění v rámci moderních operačních systémů vytvořit speciální režim, který za pomocí VM86 hypervisoru běžícího v privilegovaném režimu umožňuje reálný mód virtualizovat.

9.1.1 Vstup a opuštění VM86

Hypervisor musí pro každý VM86 virtuální stroj spravovat speciální TSS (Task State Segment), který obsahuje informace jako hodnoty segmentových a obecných registrů, registr příznaků, ukazatel instrukcí, řídící registr CR3 (fyzickou adresu stránkovací tabulky nejvyšší úrovně pro převod lineární adresy v rámci VM86 na adresu fyzickou) a bitovou mapu I/O portů, na které může kód běžící ve VM86 přímo přistupovat.

Přepnutí do režimu VM86 je možné pomocí instrukce JMP nebo CALL směrující na příslušný TSS nebo bránu na něj mířící. TSS definující VM86 má v registru příznaků nastaven bit VM. Další možností je návrat z přerušení instrukcí IRET, které opět nastaví bit VM registru příznaků.

Opustit režim VM86 lze několika způsoby. Nejčastěji se tak stane při příchodu hardwarového přerušení, kdy je řízení předáno operačnímu systému v privilegovaném režimu (obvykle se poté dojde k běžné obsluze hardwaru a později k návratu do VM86). Další možností je výjimka způsobená VM86 kódem, případně pokus o vykonání některé z instrukcí CLI, STI, PUSHF, POPF, IN, OUT, INS, OUTS, HLT, INT nebo IRET za jistých podmínek, kdy dojde k vyvolání výjimky General Protection Fault, řízení získá opět operační systém běžící v privilegovaném režimu a může tak zavolat hypervisor pro ošetření vzniklé situace.

9.1.2 Virtualizace paměti

Z pohledu virtualizovaného prostředí poskytuje VM86 možnost přistupovat k 65536 překrývajícím se segmentům paměti, každý o velikosti 65536 bytů. Výsledná lineární adresa se vypočítá z čísla segmentu a offsetu v rámci něj podle vztahu

$$\text{linear} = 16 \times \text{segment} + \text{offset}. \tag{9.1}$$

Tato lineární adresa se převádí na fyzickou adresu standardním mechanismem stránkování. Emulovat tak lze jak chování originální čistě 20bitové datové sběrnice, kdy adresy nejvyššího segmentu 0xFFFF počínaje offsetem 0x0010 přetékaly na fyzické adresy 0x0000 až 0xFFEF, tak chování volitelné od procesoru 80286, kdy bylo možné díky 24bitové datové sběrnici využít oněch 65520 bytů navíc.

VM86 hypervisor se obvykle nachází ve stejném adresovém prostoru na adrese vyšší než 0x10FFEF a má tedy snadný přístup do paměti virtualizovaného reálného módu.

9.1.3 Virtualizace I/O operací

Přístup k I/O portům je ovlivněn nastavením bitmapy povolených portům v TSS. Obvykle nemá VM86 kód přístup k žádným portům (jen zcela výjimečně se povoluje exkluzivní přístup k nějaké konkrétní periférii) a při každém takovém pokusu dochází k vyvolání výjimky a předání obsluhy hypervisoru, který poté provede simulaci přístupu k odpovídající virtuální periférii.

9.1.4 Virtualizace přerušení

Přerušení vzniklá v průběhu zpracování VM86 kódu se dělí do tří kategorií a každá se se zpracovává specifickým způsobem.

Softwarové přerušení

Při vyvolání instrukce INT se využívá informace v mapě přesměrování přerušení v rámci TSS, kdy může dojít buď k přímému vyvolání příslušné obslužné rutiny v privilegovaném režimu, k vyvolání General Protection výjimky nebo spuštění obsluhy přímo ve VM86 podle vektoru přerušení v reálném módu.

Maskovatelná hardwarová přerušení

Pokud příslušný procesor podporuje mechanismus virtuálních přerušení, dovoluje to hypervisoru efektivněji ošetřovat případy, kdy VM86 kód zakazuje a povoluje maskovatelná hardwarová přerušení instrukcemi CLI a STI (čímž ovšem ovlivňuje stav fyzického stroje, takže hypervisor musí tyto instrukce odchytávat a implementovat interní stav nedoručování přerušení VM86 virtuálnímu stroji).

Mechanismus virtuálních přerušení umožňuje, aby instrukce CLI a STI v rámci VM86 ovlivňovaly místo standardního příznaku maskování přerušení IF jeho virtuální variantu VIF.[1]

Další speciální příznak VIP (Virtual Interrupt Pending) používá hypervisor v situaci, kdy jsou VM86 aktuálně přerušení maskována a nějaké přijde. Nastavením tohoto příznaku dojde po provedení instrukce STI v rámci VM86 okamžitě k přerušení, stejně jako by se stalo v případě reálného módu.

Hardwarová přerušení a výjimky

Při příchodu přerušení nebo výjimky, neplatí-li žádná z podmínek uvedená v odstavcích výše, dojde k přímému volání obslužné rutiny v privilegovaném režimu, která může zkontrolovat, zda před jejím spuštěním neběžel VM86 režim a podle toho případně předat řízení hypervisoru.

I tento režim umožňuje, aby hypervisor předal skutečnou obsluhu přerušení obslužné rutině ve VM86 režimu, pouze se jedná o poměrně komplikovaný postup, který vyžaduje také správné ošetření instrukce IRET na konci VM86 rutiny.

9.2 Power Hypervisor

Procesorová architektura POWER a z ní odvozená PowerPC definuje tři možné režimy práce procesoru. První dva jsou běžný privilegovaný a uživatelský režim, které jsou implementovány v každém procesoru, který vyhovuje specifikaci architektury. Volitelný třetí režim se nazývá *hypervisor* a slouží k podpoře *logického partitioningu*, což je v místní terminologii synonymum pro hardwarovou podporu virtualizace.

Přístup do paměti je na architektuře POWER a PowerPC možný dvěma způsoby: se zapnutým překladem virtuálních adres na adresy fyzické (použit je mechanismus stránkování s jednoúrovňovou

[1] Také při čtení celého registru příznaků pomocí instrukce PUSHF je místo bitu IF nepozorovaně použit bit VIF.

hashovanou stránkovací tabulkou), nebo v takzvaném reálném módu, kdy virtuální adresy odpovídají přímo adresám fyzickým (v tomto módu jsou například prováděny obslužné rutiny přerušení).

Technologie Power Hypervisor rozšiřuje tento paměťový model o dva registry RMOR (real mode offset register) a RMLR (real mode limit register), které definují bázovou adresu a maximální velikost fyzické paměti, kterou může kód běžící v privilegovaném nebo uživatelském režimu a v obou módech přístupu do paměti použít. Tyto registry jsou pochopitelně přístupné pouze v režimu hypervisor a slouží tedy k virtualizaci fyzické paměti.

Pomocí řídícího registru LPES se v režimu hypervisor nastavuje, zda jsou přerušení ošetřována kódem běžícím v privilegovaném režimu, nebo zda jsou ošetřována v režimu hypervisor. což umožňuje virtualizaci přerušení. Pro implementaci hypervolání je možné přepnout se z privilegovaného režimu do režimu hypervisor také pomocí speciální varianty instrukce SC (system call).

Je-li technologie Power Hypervisor aktivní, vyvolávají veškeré instrukce, které by při svém provedení ve virtuálním stroji mohly ovlivnit stav stroje fyzického (především zápis do některých řídících registrů), přerušení Program Interrupt, které ošetřuje hypervisor. Také čtení hodnot některých řídících registrů způsobuje vyvolání této výjimky, případně jsou přečtené hodnoty automaticky maskovány tak, aby byla dodržena podmínka ekvivalence.

Procesory POWER5 podporují pro usnadnění preemptivního plánování virtuálních strojů kromě běžného *decrementer* přerušení (vnitřní časovač procesoru) také nezávislé přerušení *hypervisor decrementer*.

9.3 Vanderpool

Technologie podpory úplné virtualizace procesorů Intel platformy IA-32 a EM64T[2], označovaná též VT-x, je dostupná na pozdějších řadách procesorů Pentium 4, Pentium D, Xeon a Core Duo.[3] Smyslem tohoto rozšíření není možnost triviálně implementovat virtualizaci všech periférií PC (k tomu je stále potřeba naprogramovat v hypervisoru vlastní správu zdrojů jako jsou sběrnice, paměťově mapovaná zařízení a DMA, případně využít k těmto účelům prostředky hostitelského operačního systému), umožňuje však, aby platformy IA-32 a EM64T splňovaly dostačující podmínky úplné virtualizace (viz sekce 3.2.3).

Kromě stávajících režimů práce procesoru jsou zavedeny další dva v podstatě ortogonální režimy *VMX root operation* (sloužící pro běh virtualizéru), ve kterém jsou dostupné nové VMX instrukce, a *VMX non-root operation* (sloužící pro běh virtuálního stroje), ve kterém je dostupná nová privilegovaná instrukce VMCALL sloužící pro implementaci hypervolání. To, že se procesor nachází v režimu non-root operation, nelze rozlišit žádným příznakem.

Technologie VT-x se aktivuje nastavením příslušného bitu v řídícím registru CR4 a vykonáním instrukce VMXON, jejíž argument je fyzická adresa rámce obsahujícího strukturu *VMXON region* (procesor ji používá pro udržování informací podstatných pro realizaci virtualizace). V rámci režimu root operation lze VT-x deaktivovat instrukcí VMXOFF.

9.3.1 Virtual Machine Control Structure

Každý virtuální stroj je řízen daty uloženými ve struktuře VMCS. Aktuální struktura VMCS se nastaví, resp. přečte v režimu root operation pomocí instrukce VMPTRST, resp. VMPTRLD (určena je fyzickou adresou rámce, který strukturu obsahuje). Deaktivaci aktivní VMCS lze provézt instrukcí VMCLEAR.

[2] Drobně modifikovaná varianta platformy AMD64 definovaná společností Intel.

[3] Stejné označení nese také podobná technologie pro platformu IA-64.

Virtuální stroj definovaný aktuální VMCS se spustí (tj. dojde k přechodu do non-root operation režimu) pomocí instrukce VMLAUNCH. Běh non-root operation režimu může být ukončen a k přechodu zpět do root operation režimu může dojít vykonáním instrukce VMCALL nebo v důsledku jiné události definované VMCS. Důvod takového přechodu se zjistí z hodnoty *Exit Info* a po ošetření nastalé situace hypervisorem může být virtuální stroj opět spuštěn instrukcí VMRESUME.

Jednotlivé položky VMCS se čtou a zapisují nepřímo pomocí instrukcí VMREAD a VMWRITE (jejich argumentem jsou konstanty specifikující příslušnou položku), jejich paměťová reprezentace není dokumentována.

Některé podstatné položky VMCS a jejich obsah:

- **Stav virtuálního stroje**
  Obsah registrů CR0, CR3, CR4, DR7, RSP, registru příznaků, selektor, báze a limit pro CS, SS, DS, ES, FS, GS, LDTR a TR, přístupová těchto segmentů, báze a limit pro GDTR, IDTR. Dále jsou zde uvedeny dva podstavy virtuálního stroje:
  - **Stav virtuálního procesoru**
    Aktivní – procesor vykonává instrukce
    HLT – procesor byl zastaven instrukcí HLT
    shutdown – procesor byl zastaven z důvodu trojnásobné výjimky nebo jiného problému
    wait-for-startup-IPI – aplikační procesor dosud nebyl nastartován
  - **Přerušitelnost virtuálního procesoru**
    STI – jsou maskována přerušení instrukcí STI
    MOV SS – jsou maskována přerušení z důvodu provádění instrukce MOV SS
    NMI – je maskováno nemaskovatelné přerušení
- **Stav hypervisoru**
  Obsah registrů CR0, CR3, CR4, RSP, RIP, selektor a báze pro FS, GS, TR, GDTR, IDTR, selektor pro CS, SS, DS, ES, FS, GS a TR (ostatní hodnoty jsou při vstupu do root operation režimu nastaveny na pevné konstanty).
- **Příznaky řízení virtuálního stroje**
  Sada příznaků určující, při jakých událostech dojde k opuštění non-root operation režimu a vyvolání root operation režimu. Volitelně se může jednat o vyvolání hardwarového nebo softwarového přerušení (lze specifikovat i konkrétní vektor přerušení), nemaskovatelného přerušení, vykonání instrukce HLT. INVLPG, MWAIT, RDPMC, RDTSC, MONITOR, PAUSE, nastavení nebo čtení registru CR8, čtení nebo zápis I/O portu atd.
- **I/O bitová mapa A/B**
  Fyzická adresa dvou rámců, které obsahují bitovou mapu I/O portů, při jejichž čtení nebo zápisu dojde k opuštění non-root operation režimu a vyvolání root operation režimu.
- **Offset čítače *time-stamp***
  Hodnota, která upravuje virtualizovanou hodnotu procesorového čítače *time-stamp*.
- **Masky CR0 a CR4**
  Masky bitů řídících registrů CR0 a CR4, které může virtuální stroj nastavit bez vyvolání root operation režimu.
- **Stínové hodnoty CR0 a CR4**
  Virtualizované hodnoty řídících registrů CR0 a CR4.
- **Povolené hodnoty CR3**
  V aktuální implementace VT-x až čtyři hodnoty registru CR3, které může virtuální stroj nastavit, aniž by došlo k vyvolání root operation režimu.
- ***Exit Info***
  Informace o důvodu opuštění non-root operation režimu a vyvolání root operation režimu.

9.4 Pacifica

Konkurenční virtualizační technologie společnosti AMD pro platformu AMD64, dostupná na novějších modelech procesorů Athlon 64 a Turion 64. Přestože není kompatibilní s technologií Vanderpool, její principy a způsob fungování je velmi podobný.

Mezi zajímavé vlastnosti patří například označování položek v TLB identifikátorem adresního prostoru (což urychluje přepínání mezi jednotlivými virtuálními stroji), podpora speciálního hardwarového modulu pro provádění důvěryhodného kódu nebo širší spektrum událostí, kdy dochází k vyvolání hypervisoru (přepínání úlohy, čtení a zápis všech řídících registrů atd.).

Kapitola 10

Srovnání vlastností

Asi neexistuje jediný klíč na srovnání jednotlivých virtualizačních metod, hodnotit je potřeba více kritérií, jejichž výsledná váha závisí na účelu nasazení virtualizace, přičemž některá kritéria jako například režie a míra izolace mohou být vzájemně v protikladu.

10.1 Režie

Je intuitivně zřejmé, že každá run-time abstraktní vrstva softwaru představuje zvýšení režie způsobené nenulovým počtem strojových instrukcí, které je potřeba provézt na přenos libovolné informace mezi jednotlivými koncovými rozhraními této vrstvy. V případě virtualizace tomu není jinak, jaká je ovšem skutečná režie? Kromě teoretických úvah se v této sekci podíváme také na srovnání některých skutečných produktů v testech.

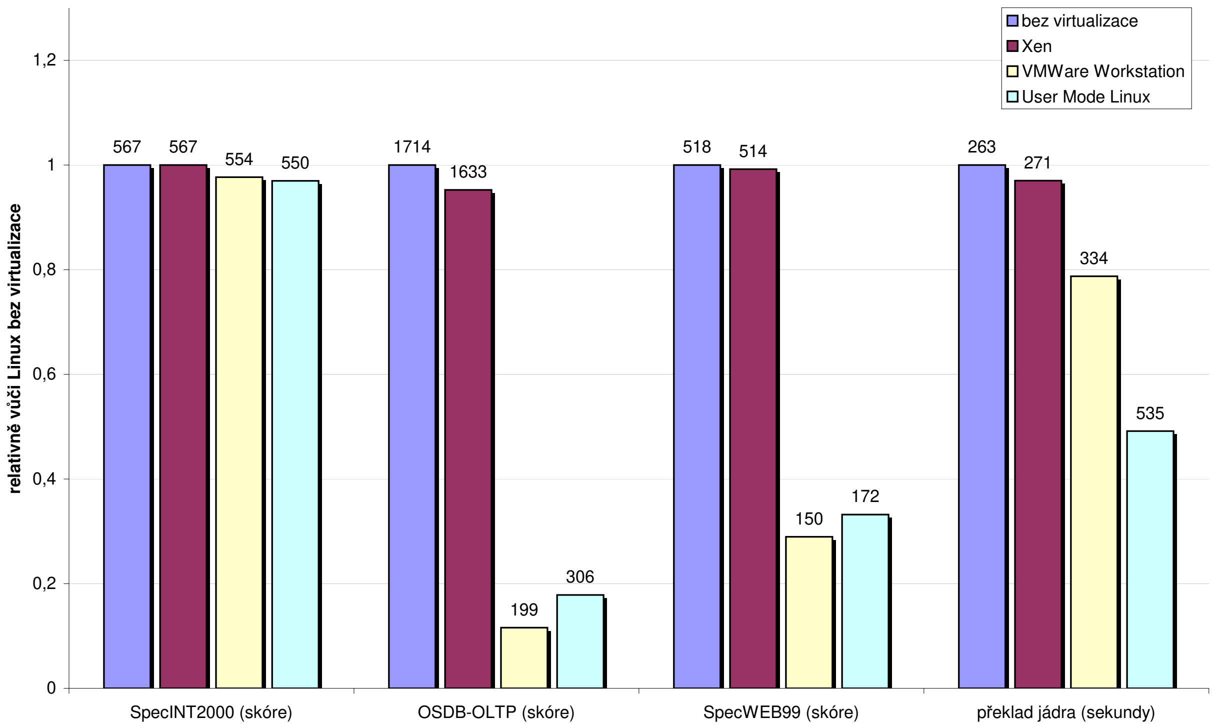
Existuje přímá úměra mezi režií jednotlivých metod virtualizace a poměrným počtem instrukcí, které daná metoda nedovede provádět nativně. Z definice se při simulaci žádná instrukce virtuálního stroje neprovádí na fyzickém stroji nativně, režie zde je tedy nejvyšší. Skutečná hodnota se poté liší podle toho, zda simulátor virtuální instrukce interpretuje nebo provádí dynamický překlad, jak dobře je optimalizován atd.

Při úplné virtualizaci a paravirtualizaci je poměrný počet instrukcí prováděných nativně přibližně shodný, o režii tedy rozhoduje převážně rychlost simulace privilegovaných instrukcí, čas spotřebovaný na ošetření nevirtualizovatelných stavů u úplné virtualizace (například na platformě IA-32), resp. složitost rutin paravirtualizéru realizujících privilegované operace. Jak demonstruje srovnání na obrázcích 10.1 a 10.2, neexistuje v tomto srovnání zcela obecný trend – ve velkém počtu případů má paravirtualizace režii menší než úplná virtualizace (obzvláště na platformách nesplňujících dostatečné podmínky úplné virtualizace), vliv kvality implementace nemá ovšem nezanedbatelný vliv.

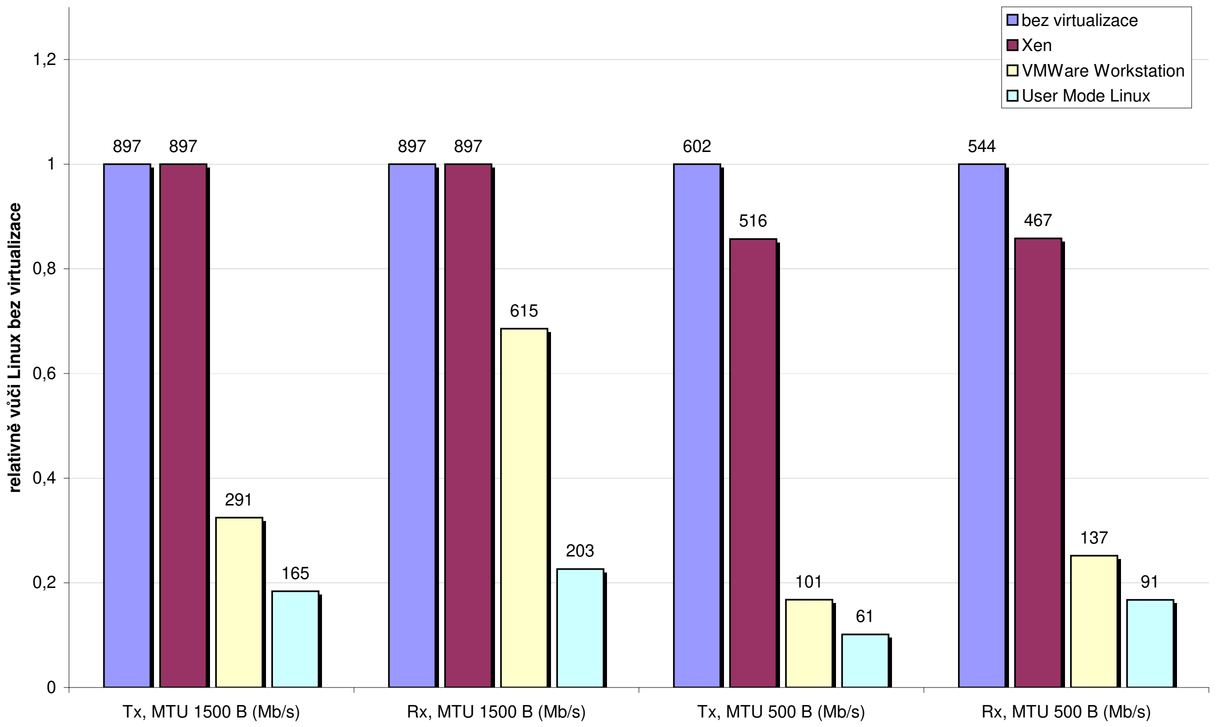
Relativní srovnání paravirtualizace a partitioningu v režii (viz obr. 10.3), kterou zvyšují latenci systémových volání jádra operačního systému, má naopak jednoznačný výsledek – partitioning způsobuje u relevantních systémových volání mnohem menší režii, protože není nijak ovlivněn nativní mechanismus správy paměti a provádění privilegovaných instrukcí, pouze se provádí několik málo testů na viditelnost entit jednotlivých kontextů.

10.2 Míra izolace

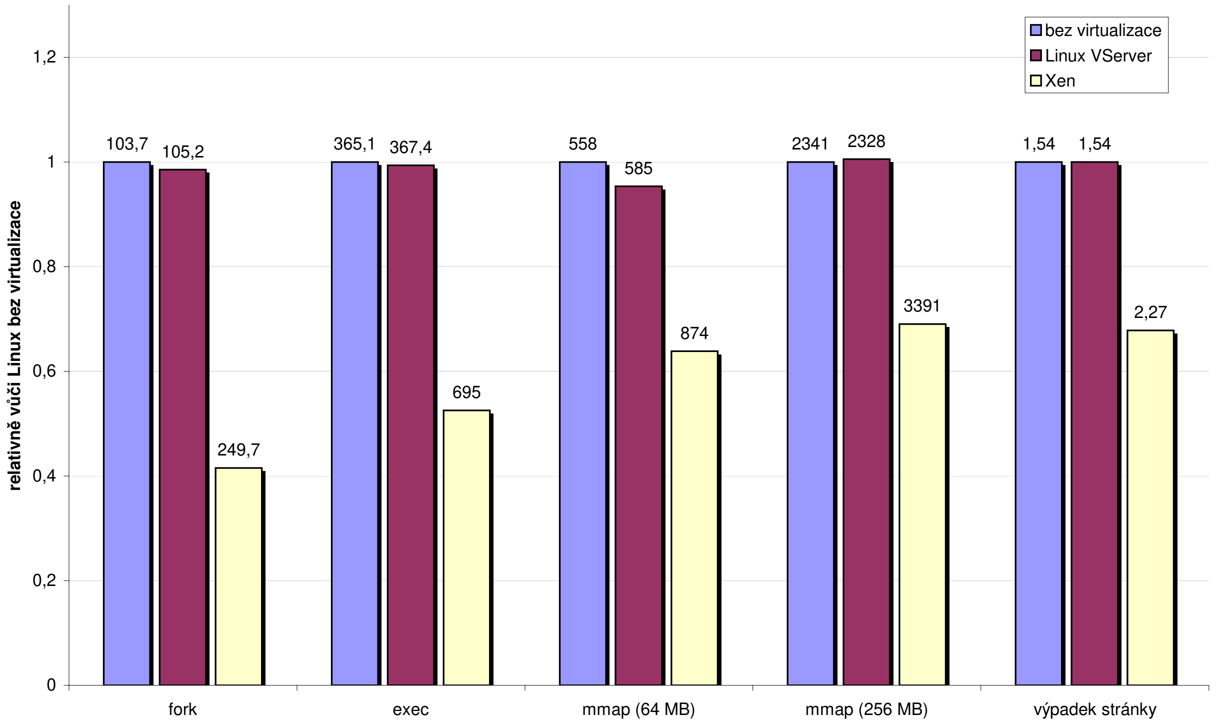
Z čistě teoretického pohledu by měla být míra izolace jednotlivých virtuálních strojů všemi metodami shodná, neboť program běžící ve virtuálním stroji by neměl nijak ovlivňovat běh jiných virtuálních strojů nebo dokonce stroje fyzického. Jakékoliv porušení izolace lze považovat za závažnou chybu implementace virtualizace.



Obr. 10.1: Srovnání výkonu tří standardních benchmarků a doby shodného překladu zdrojových kódů jádra Linuxu (zdroj [3]).



Obr. 10.2: Srovnání datové propustnosti TCP stacku při přijímání (Tx) a vysílání (Rx) paketů udané velikosti (MTU) (zdroj [3]).



Obr. 10.3: Microbenchmark lmbench, latence systémových volání v mikrosekundách (zdroj [6]).

Zkoumejte tedy spíše možné důsledky podobné chyby. U simulátoru představuje chyba izolace virtuálního stroje možnost pádu procesu simulátoru v rámci hostitelského operačního systému, což je ovšem záležitost, která v moderních operačních systémech neznamená žádné bezpečnostní riziko pro ostatní procesy. U partitioningu představuje největší nebezpečí ta situace, kdy porušení izolace virtuálního stroje vede k tomu, že superuživatel virtuálního stroje získá možnost komunikovat se superuživatelskými procesy stroje fyzického (resp. toho kontextu, který má privilegia ovlivňovat konfiguraci fyzického stroje).

Naproti tomu virtualizér a paravirtualizér přímo ovlivňují privilegované stavy fyzického stroje, proto může podobné porušení izolace vézt nejčastěji k zhroucení nebo zaseknutí fyzického stroje.

10.3 Bezpečnost

Virtualizace přináší kromě klasického pohledu na bezpečnost jako důsledek izolace ještě nový, zcela nečekaný pohled. Operační systém totiž při ideálním způsobu fungování virtualizace nemůže vůbec detekovat, že ve skutečnosti neběží na fyzickém stroji. Tato pozitivní vlastnost se stává problémem, pokud existuje cesta, jak systém „unést“ a spustit ve virtuálním stroji, aniž by to byl záměr jeho správců. Veškeré bezpečnostní mechanismy operačního systému by se tak okamžitě staly bezzubými, protože hypervisor má úplnou informaci o stavu virtuálního stroje.

Všechny hardwarové technologie pro usnadnění virtualizace pochopitelně pouze dovolují, aby se do režimu hypervisoru přepnul kód běžící v privilegovaném režimu procesoru, nikoliv tedy uživatelský kód. Bohužel libovolná bezpečnostní chyba operačního systému, která umožní spustit jen několik málo privilegovaných instrukční vytvořených útočníkem, umožní nejen získat nad systémem plnou kontrolu, ale dokonce kompromitovat systém efektivně nedetekovatelným způsobem.

Tento aspekt počítačové bezpečnosti není v současné době prakticky vůbec prozkoumán a existuje jen několik málo proof-of-concept studií. Jako možná ochrana proti podobnému druhu útoku se jeví například to, že systém bude stále virtualizován a veškeré pokusy o spuštění rekurzivní virtualizace budou hlídány a hlášeny.

10.4 Determinismus

Zcela deterministický a opakovatelný způsob provádění kódu virtuálního stroje poskytují pouze simulátory, které pracují na principu diskrétní simulace. Virtualizéry, paravirtualizéry i partitioning díky tomu, že provádějí část svého kódu nativně a jejich virtuální hardware je přímo ovlivňován fyzickým hardwarem, neumožňují dosáhnout snadným způsobem determinismus.

10.5 Accounting

Měření výpočetního času spotřebovaného virtuálními stroji souvisí se způsobem provádění virtuálních instrukcí. U simulátorů můžeme snadno změřit počet cyklů virtuálního procesoru, u ostatních metod jsme odkázáni na nepřímé hodnoty odvozené z časových razítek fyzického procesoru, případně přerušení časovače. V případě, že hostující operační systém tyto údaje sám sbírá, je možné je přímo použít, není ovšem vždy snadné rozlišit čas samotného provádění a simulace virtuálních instrukcí od času způsobeného jinou režií.

Kapitola 11

Virtualizace systému HelenOS

HelenOS je multiplatformový vývojový operační systém, který vzniká na MFF UK od roku 2005 jako rozšíření původního jádra SPARTAN Jakuba Jermáře (jehož samostatný vývoj probíhal v letech 2001 až 2004). Veškeré informace o systému HelenOS, včetně zdrojových kódů šířených pod licencí BSD a dokumentace, je možné získat na webu `http://www.helenos.eu/`.

11.1 Struktura systému HelenOS

Operační systém HelenOS je inspirován koncepcí mikrojádra – důraz je kladen především na komunikaci mezi úlohami (IPC) a možnost realizovat většinu ovladačů hardwaru a dalších systémových funkcí jako služby poskytované úlohami běžícími v uživatelském režimu procesoru, zatímco samotné jádro běžící v privilegovaném režimu má na starosti především správu fyzické a virtuální paměti, podporu hardwaru nejnutnějšího pro běh systému (řadič přerušení atd.), plánování úloh a několik dalších nízkoúrovňových záležitostí.

Kód jádra SPARTAN je logicky rozdělen do tří částí: na platformově nezávislou část (která je společná pro všechny podporované platformy, definuje chování jádra a jeho API vůči uživatelským úlohám), platformově sdílenou část (kód používaný několika, ale ne nutně všemi platformami – podpora jaderného framebufferu pro textový výstup, různé instance mechanismů mapování virtuální paměti na fyzickou atd.) a platformově závislé části (implementující některé funkce vnitřního API, které nelze abstrahovat nezávisle na platformě). Jako doplněk těchto částí obsahuje strom jádra regresní testy a další doplňky určené pro vývoj, nikoliv pro samotný běh.

Funkčně lze jádro SPARTAN rozdělit na několik subsystémů:[1]

ADT Implementace datových struktur používaných v jiných částech jádra (spojové seznamy, hashovací tabulky, bitové mapy, B+ stromy).
boot Převážně platformě závislá infrastruktura pro spuštění jádra a nastavení prostředí, ve kterém může běžet platformově nezávislý kód.
CPU Správa procesorů.
DDI Rozhraní sloužící pro správu hardwarových zdrojů a API umožňující implementovat ovladače zařízení jako úlohy v uživatelském režimu (mapování paměti zařízení do adresního prostoru úlohy, povolování přístupu k I/O portům, převod externích přerušení na IPC notifikace).
drivers Ovladače zařízení nezbytně nutné pro samotný běh jádra.
interrupt Univerzální mechanismus ošetřování výjimek a přerušení.
IPC Jaderná část funkcí realizující komunikaci mezi úlohami pomocí zasílání zpráv.

[1] Rozdělení subsystémů zhruba odpovídá podadresářům adresáře `generic/src`, resp. `arch/`*`platforma`*`/src`.

kconsole Interaktivní ladící nástroj, který umožňuje zjišťovat různé běhové parametry jádra a provádět introspekci. Součástí jeho kódu je také vnitřní API pro jaderné ovladače vstupních a výstupních znakových zařízení. Jiné subsystémy mohou registrovat příkazy kconsole.
library Knihovna pomocných funkcí, například parser hlaviček binárního formátu ELF.
main Základní část jádra starající se o jeho inicializaci a rutinní běh.
MM Správa fyzické paměti (rozdělení dostupné fyzické paměti do zón rámců, buddy alokátor nad těmito rámci, vnitřní API pro mapování fyzické paměti do nezávislých virtuálních adresních prostorů, rozhraní pro správu TLB atd.).
proc Subsystém realizující správu úloh, vláken a plánování.
security Část jádra přidělující oprávnění speciálním úlohám, například ovladačům zařízení nebo superprivilegované úloze (která poté realizuje bezpečnostní politiku).
SMP Podpora strojů s více symetrickými procesory.
synch Implementace aktivních (spinlock) a pasivních synchronizačních primitiv (čekací fronta vláken, semafor, mutex, futex, podmínková proměnná, RW-zámek).
syscall Vrstva systémových volání, která zprostředkovává veřejné API jádra vůči uživatelským úlohám.
sysinfo Hierarchická databáze běhových informací jádra.
time Funkce starající se o události, které jsou naplánovány v čase (např. timeout synchronizačních primitiv).

11.2 Implementace partitioningu

Při vytváření zadání diplomové práce padla v případě praktické implementace volba na partitioning, protože umožňuje vsunout virtualizační vrstvu do platformově nezávislých částí jádra operačního systému a tak dosáhnout dokonalé přenositelnosti.

Díky přehledné struktuře systému HelenOS, kdy jeho mikrojádro provádí pouze velmi omezenou část činností a většina operační logiky je přesunuta do úloh běžících v uživatelském režimu procesoru, je implementace kontextů velmi jednoduchá a přímočará. Každý kontext je v jádře identifikován číselnou hodnotou typu `context_id_t` (definovanou v souboru `kernel/generic/include/typedefs.h`.

Kontext entit po bootu systému je určen konstantou `DEFAULT_CONTEXT` ze souboru `kernel/generic/include/arch.h`. Základními entitami, které jsou rozděleny do kontextů, jsou úlohy.[2] Struktura `task_t` (resp, struct task) definující úlohu (soubor `kernel/generic/include/proc/task.h`) je tedy rozšířena o položku `context_id_t context`.

V souboru `kernel/generic/include/arch.h` je definováno makro pro základní test, zda jsou dva kontexty shodné:

```
#define context_check(ctx1, ctx2)   ((ctx1) == (ctx2))
```

Tato základní podoba testu je zcela symetrická, protože všechny kontexty si jsou zcela rovnocenné. Případné zavedení *spectator contextu* je ovšem velmi snadné, stačí do makra `context_check` přidat test, zda argument `ctx2` je identifikátor tohoto speciálního kontextu (zároveň je potřeba modifikovat sémantiku tohoto makra, protože jeho argumenty tak přestanou být symetrické).

Jádro definuje aktuálně běžící úlohu pomocí struktury `THE`[3], která je uložena na vrcholu aktuálního jaderného zásobníku a obsahuje tedy odkaz na úlohu – hodnota její položky `context` definuje aktuální kontext.

Kontextový test je použit na těchto místech kódu jádra:

[2] V terminologii HelenOS synonymum pro procesy v unix-like operačních systémech.
[3] Task, tHread, Executing CPU.

- **Funkce `ddi_iospace_enable`**
  Soubor `kernel/generic/src/ddi/ddi.c`
  Tato funkce implementuje systémové volání `SYS_IOSPACE_ENABLE`, pomocí kterého privilegovaná úloha (správce zařízení) umožňuje jiné úloze přístup k I/O portům (na platformách, kde existuje nezávislý prostor I/O portů). Je potřeba zajistit, aby nemohla změnit možnost přístupu úloze z jiného kontextu.
- **Funkce `sys_cap_grant`**
  Soubor `kernel/generic/src/security/cap.c`
  Tato funkce implementuje systémové volání `SYS_CAP_GRANT`, pomocí kterého superprivilegovaná úloha (správce oprávnění) nastavuje jiné úloze oprávnění (například proto, aby se stala správcem zařízení). Test zajišťuje, že superprivilegovaná úloha může změnit oprávnění úloze z jiného kontextu.
- **Funkce `task_create`**
  Soubor `kernel/generic/src/proc/task.c`
  Funkce vytváří novou úlohu a běží-li již úloha Naming Service, nastavuje nové úloze iniciální IPC spojení na tuto úlohu implementující jmennou službu. Kontroluje se zde, zda je jmenná služba ze stejného kontextu jako nově vznikající úloha. Vzhledem k tomu, že nová IPC spojení lze navazovat jen přes iniciální spojení na jmennou službu, zajišťuje tento test to, že žádná úloha nemůže poslat IPC zprávu úloze mimo aktuální kontext.

11.3 Dílčí závěr

Zavedení základní infrastruktury pro partitioning do jádra systému je záležitost velmi snadná, myšlenkově nejnáročnější část je správně identifikovat konkrétní místa zdrojového kódu, kde dochází k interakci jednotlivých entit a kde je potřeba přidat kontextové testy pro dosažení požadované izolace.

11.4 Portování na platformu Xen

V době vypracování této diplomové práce se ukázalo vhodné implementovat nad rámec původního zadání také ukázku úpravy jádra pro běh na virtuálním stroji definovaném paravirtualizérem. Jako nejvhodnější kandidát byl zvolen paravirtualizér Xen (konkrétně 32bitová varianta pro platformu IA-32, bez podpory PAE), který má širokou podporu komunity svobodných operačních systémů i komerčních společností.

Podpora pro paravirtualizaci i partitioning je na sobě zcela nezávislá, takže je možné obě metody kombinovat a získat tak „dvojitou virtualizaci“.

Jako výchozí platforma pro vytvoření nového portu jádra SPARTAN byla zvolena jaderná architektura *ia32* (obsahující podporu SMP strojů IA-32). Nová architektura byla pojmenována *xen32* a s původní *ia32* sdílí velkou část kódu. Protože kód vykonávaný v uživatelském režimu zůstává i v rámci paravirtualizace shodný, žádné změny se netýkají uživatelské architektury *ia32*.

Následující sekce obsahují převážně popis změn, které bylo potřeba mezi oběma architekturami udělat, a doplňuje popis fungování Xenu z 7.1.

11.4.1 Bootování

Drobné změny se týkají procesu bootování a binárního formátu jádra. Systém HelenOS bootuje tak, že podporovaný boot loader načte do fyzické paměti obraz jádra (na některých platformách přímo na vhodnou fyzickou adresu, na jiných provede jádro svou vlastní relokaci) a obrazy nejpodstatnějších iniciálních uživatelských úloh, které slouží pro vytvoření běhového prostředí (Naming Service, podpora IPC, základní ovladače zařízení atd.).

Jako boot loader je stále i pro *xen32* použit GNU GRUB, ale jako primární image je použito mikrojádro Xen (3.0.2) a teprve jako jeho modul je použito jádro systému HelenOS – SPARTAN. Iniciální úlohy jsou načteny jako další moduly.

Formát jádra se oproti architektuře *ia32* drobně liší. Místo binárního formátu s multiboot hlavičkou je výsledná podoba slinkovaného jádra ELF. Odstraněna byla sekce `unmapped`, která byla linkována na virtuální adresy těsně za prvním megabytem paměti, protože jádro je Xenem rovnou načteno a spuštěno na požadované virtuální adrese 0x80000000 (2 GB).

Naopak do ELF obrazu byla přidána sekce `__xen_guest` obsahující tyto informace o jádře:

- GUEST_OS=HelenOS
  Pojmenování systému.
- XEN_VER=xen-3.0
  Verze rozhraní hypervisoru, které jádro požaduje.
- HYPERCALL_PAGE=0x0000
  Nultá stránka virtuálního adresního prostoru jádra (tedy adresy 0x80000000 až 0x8000FFFF) je vyhrazena pro hypervolání. Tento kód do stránky označené symbolem `hypercall_page` vsune Xen.
- LOADER=generic
  Pro načtení jádra bude použit standardní zavaděč.
- FEATURES=writable_page_tables
  Jádro bude používat zapisovatelné stránkovací tabulky.

Kromě stránky `hypercall_page` jsou na začátku obrazu jádra vyhrazeny další dvě stránky. Stránka `shared_info` je určena pro namapování fyzického rámce obsahujícího strukturu stejného názvu (ta obsahuje informace o virtuálních procesorech a kanálech událostí) a stránka `console_page` bude sloužit pro namapovaní fyzického rámce pro vstupně/výstupní konzoli (toto přemapování umožní zachovat standardní 1:1 identické mapování adresního prostoru jádra).

11.4.2 Inicializace

Vstupní bod jádra, funkce `kernel_image_start` (soubor `kernel/arch/xen32/src/boot/-boot.S`), provede nejprve zkopírování struktury `start_info_t`, která se nachází za tabulkou P2M (tedy v oblasti paměti, kterou budeme chtít v budoucnu uvolnit), do statické struktury v jádře. Podobně je přepnut zásobník na dočasný zásobník v rámci jádra, aby bylo možno bootovací zásobník uvolnit pro stránkovací tabulky.

Další inicializace probíhá ve funkci `arch_pre_main()` (soubor `kernel/arch/xen32/-src/xen32.c`), kde dojde nejprve k zapnutí podpory zapisovatelných stránkovacích tabulek, přemapování `shared_info` a `console_page` (viz výše) a vytvoření 1:1 identického mapování adresního prostoru jádra na všechny volné fyzické rámce, které má jádro od hypervisoru k dispozici (mapování je 1:1 identické v tom smyslu, že čísla stránek odpovídají s posunem o 0x80000 číslům abstraktních rámců, nikoliv fyzických rámců).

Další inicializace probíhá již standardně v rámci platformově nezávislé části jádra.

11.4.3 Správa fyzické paměti

Správa fyzické paměti (`kernel/arch/xen32/src/mm/frame.c`) je inicializována s jedinou zónou paměti, která obsahuje všechny abstraktní rámce počínaje prvním rámcem za iniciálním stránkovacím adresářem (pt_base). Z této zóny jsou však již některé rámce použity pro stránkovací tabulky nižších úrovní pro identické mapování, ty jsou tedy označeny jako nedostupné.

Kromě toho jsou zavedena makra `PA2MA` a `MA2PA` pro převod adresy vyjádřené v soustavě abstraktních rámců na adresu v soustavě rámců fyzických a zpět. Pro převod prvním směrem je použita tabulka P2M ze struktury `start_info_t`, pro převod druhým směrem globální tabulka M2P namapovaná na adrese 0xFC000000.

Jádro používá „ploché“ segmenty přednastavené Xenem, nemusí tedy používat hypervolání pro nastavení vlastních segmentů.

11.4.4 Správa virtuální paměti

Pro správu mapování (`kernel/arch/xen32/src/mm/page.c`) je stejně jako v případě architektury *ia32* použita instance mapovacího rozhraní `page_pt` (obecné hierarchické 4úrovňové stránkovací tabulky). Parametry instance zůstávají shodné jako pro fyzický stroj IA-32, tedy stránkování je efektivně dvouúrovňové, obě úrovně stránkovacích tabulek obsahují 1024 položek a velikost rámce/stránky je 4 KB.

Čtení položek stránkovacích tabulek se provádí přímo, také se provádí zápis do tabulek nižší úrovně (využívá se rozšíření Xenu – zapisovatelné stránkovací tabulky). K zápisu položek do stránkovacího adresáře je použito hypervolání MMU_UPDATE a ke změně fyzické adresy stránkovacího adresáře hypervolání MMUEXT_OP. Speciálně je třeba ošetřit, aby položky stránkovacího adresáře měly správné příznaky a stránkovací tabulky nižší úrovně nebyly označeny jako zapisovatelné.

11.4.5 Správa TLB

Ke správě TLB záznamů (invalidace položek) se používají hypervolání MMUEXT_OP s příslušnými parametry, viz `kernel/arch/xen32/src/mm/tlb.c`.

11.4.6 Výstup na konzoli

Způsob výstupu znaků na konzoli se liší podle toho, zda je jádro spuštěno jako Dom0 nebo DomU. V prvním případě je použito hypervolání CONSOLE_IO_WRITE (zápis do ladící konzole), protože virtuální konzole není v Dom0 k dispozici.

V DomU lze použít zápis do stránky `console_page`, po každém zápise dochází k notifikaci Xenu kanálem událostí `console_evtchn`. Implementace je v souboru `kernel/arch/xen32/src/drivers/xconsole.c`.

11.4.7 Přerušení a výjimky

Povolování a zakazování přerušení se děje pomocí nastavování příslušných virtuálních příznaků ve struktuře `shared_info` (inline funkce v `kernel/arch/xen32/include/asm.h`).

Nastavení obslužných rutin přerušení probíhá tak, že pomocí hypervolání SET_TRAP_TABLE je nastavena každému používanému vektoru přerušení obslužná rutina `trap()` (`kernel/arch/xen32/src/pm.c`), která se stará o předání informací standardní platformově nezávislé rutině pro obsluhu přerušení `exc_dispatch()`. Pro přehlednost jsou jednotlivé obsluhy výjimek namapovány stejně jako v případě architektury *ia32*.

11.5 Shrnutí

Z implementace podpory pro Xen lze vyvodit několik závěrů:

- Virtuální stroj definovaný paravirtualizérem je skutečně velmi podobný odpovídajícímu fyzickému stroji, při vhodně navrženém rozhraní mezi platformově nezávislými a platformově závislými částmi nepředstavují změny v kódu velký objem.
- Dokumentace API Xenu je velmi stručná a nepřesná, mnohdy jsou zcela opomenuty některé podstatné detaily, což vede k nutnosti postupovat metodou pokusů a omylů. Důsledkem je zbytečné snižování stability virtualizovaných systémů.
- Rozhraní používaná Xenem, včetně různých datových struktur, jeví známky překotného vývoje a občas značné inkonzistence (kdy jsou analogické věci řešeny zcela rozdílným způsobem). Také ze zdrojového kódu Xenu je patrné, že by bylo vhodné jej důkladně pročistit.

K této diplomové práci je přiloženo live CD – speciální modifikace periodicky vydávaného live CD systému HelenOS, obsahující přímo spustitelnou podobu systému HelenOS pro platformy IA-32 a AMD64 a vývojové prostředí v GNU/Linuxu. Demonstruje běh systému HelenOS pod Xenem jako Dom0 (výběrem přímo z bootovacího menu live CD) a také jako DomU (spuštění systému HelenOS v novém virtuálním stroji z Dom0 prostředí GNU/Linuxu).

Implementace podpory Xenu v systému HelenOS není pochopitelně vyčerpávající, chybí podpora pro frontend ovladače zařízení i privilegované operace hypervisoru pro spouštění nových virtuálních strojů. Přesto demonstruje základní funkčnost a umožňuje provozovat HelenOS ve virtuálním stroji paralelně vedle jiných systémů.

Kapitola 12

Závěr

Virtualizace běhu operačních systémů představuje proces definování virtuálního stroje nad strojem fyzickým, což poskytuje rozšíření dosavadních bezpečnostních modelů operačních systémů a to buď o izolaci skupiny procesů v rámci jednoho systému (partitioning) nebo o izolaci celých operačních systémů (další popisované metody). Výhody tohoto přístupu jsou především: snadnější administrace těchto systémů (včetně možnosti delegování práv původního superuživatele), zvýšení odolnosti proti útokům, efektivní využití hardwarových zdrojů a výpočetního výkonu (jejich sdílení mezi virtuálními stroji), větší škálovatelnost, odolnost vůči nestabilitám jednotlivých virtuálních strojů, možnost provozu starších aplikací v původním prostředí nebo vývojových verzí bez ovlivnění produkčních systémů atd.

Virtualizační přístupy lze rozdělit podle dvou základních hledisek. Je to podoba fyzického rozhraní virtualizační vrstvy (zda komunikuje přímo s fyzickým hardwarem, používá prostředky hostitelského operačního systému nebo se jedná o kombinaci těchto přístupů) a podoba metody virtualizace. Především rozlišujeme simulaci (interpretované provádění všech instrukcí virtuálního stroje), úplnou virtualizaci (interpretují se pouze privilegované instrukce, instrukce uživatelského režimu procesoru se provádí přímo), paravirtualizaci (použití privilegovaných instrukcí je v operačním systému nahrazeno explicitním voláním služeb paravirtualizéru) a partitioning (virtualizační vrstva je přímo součástí jádra operačního systému a izoluje jeho entity do samostatných kontextů).

Úplná virtualizace je metoda, která v sobě spojuje výhody rozumné malé režie (v porovnání např. se simulací) a možnosti virtualizovat běh neupravených operačních systémů. Fyzický stroj musí ovšem pro implementaci úplné virtualizace splňovat sadu nutných podmínek (detailně viz sekce 3.2.3), také jsou definovány podmínky dostačující (viz 3.2.3). Bez jejich splnění je sice možné na dané platformě úplnou virtualizaci v omezené míře provozovat, ale není možné zajistit její bezvadnou funkci za všech okolností. Nejběžnější platformy IA-32 a AMD64 samy o sobě tyto nutné podmínky nesplňují, o nápravu se snaží rozšíření instrukčních sad o technologie Vanderpool (Intel) a Pacifica (AMD).

Metody virtualizace jsou s výjimkou simulace a partitioningu silně vázány na konkrétní fyzickou a virtuální platformu a také jejich použití ve víceprocesorových systémech není vždy přímočaré.

V oblasti simulátorů existuje poměrně široká řada produktů s různými vlastnostmi a cílovým zaměřením. Mezi úplnými virtualizéry dominují produkty řady VMware, které představují nejstarší a zároveň modelovou implementaci úplné virtualizace na platformě IA-32, existují však i další zatím méně používané produkty. Zřejmě nejrozšířenějším paravirtualizérem je Xen, který je určen pro platformy IA-32 a AMD64 a je na něj portována celá řada operačních systémů. Mezi metodami implementujícími partitioning existuje značná diverzita, různé produkty pro různé operační systémy však poskytují většinou vzájemně srovnatelné základní vlastnosti.

Po srovnání virtualizačních přístupů existuje několik často protichůdných kritérií, výběr vhodné metody je tedy vždy závislý na zcela konkrétních požadavcích. Z hlediska režie spotřebované virtualizací vychází zdaleka nejlépe metoda partitioningu, ta ovšem neumožňuje paralelní běh více nezávislých operačních systémů. Dále v pořadí efektivity následuje paravirtualizace, nevýhodou této metody je ovšem to, že vyžaduje úpravu operačních systémů, jejíž běh virtualizuje. Simulace je metoda s největší režií, přesto má své uplatnění v případě, že fyzický a virtuální stroj jsou zcela rozdílné nebo pokud potřebujeme simulovat virtuální stroj co nejvěrněji.

Z pohledu praktické implementace je možné konstatovat, že implementace partitioningu je v dobře navrženém operačním systému velmi přímočará a snadná. Podobně lze říci, že portování operačního systému na virtuální stroj definovaný paravirtualizérem je záležitost srovnatelná s portováním systému na novou fyzickou platformu, výhodou je ovšem velká podobnost virtuálního stroje s daným fyzickým strojem.

Předložená diplomová práce splňuje všechny cíle, které byly vytyčeny v jejím zadání. Jejím přínosem je popsání souvislostí mezi všemi jednotlivými technologiemi, které jsou v textu uvedeny, zatímco běžně dostupná literatura se vždy zaměřuje úzce jen na jednu z nich.

Demonstrace rozšíření systému HelenOS o podporu partitioningu, která byla provedena speciálně pro tuto práci, ukazuje, jak může být implementace této efektivní metody virtualizace snadná a přímočará, je-li jádro operačního systému dobře navrženo. Oproti původnímu zadání byla implementační část diplomové práce rozšířena také o demonstraci portace systému HelenOS na virtuální platformu paravirtualizéru Xen.

Na tuto diplomovou práci by mělo přímo navazovat další rozšiřování vlastností vývojového operačního systému HelenOS a s tím spojený výzkum, týkající se například možností snapshotingu a migrace úloh, distribuovaných výpočtů nebo bezpečného provádění nedůvěryhodného kódu.

Bezprostředně by mělo následovat završení podpory paravirtualizéru Xen včetně možnosti privilegovaných operací, backendů a frontend ovladačů. Partitioning v rámci systému HelenOS by měl být rozšířen o speciální kontexty a možnost vytváření nových kontextů z uživatelských úloh. Další možné paralelní směry vývoje jsou definování virtuálního stroje v rámci systému HelenOS pro realizaci vlastní paravirtualizace a rozšíření jádra o podporu technologií Vanderpool a Pacifica pro virtualizaci běhu jiných neupravených operačních systémů na platformě IA-32 a AMD64.

Literatura

[1] Gerald J. Popek, Robert P. Goldberg: *Formal Requirements for Virtualizable Third Generation Architectures*, Communications of the ACM **17** (7) 412–421, 1974.

[2] Herbert Pötzl: *Linux-VServer Technology*, `http://linux-vserver.org/-Linux-VServer-Paper`, 2004.

[3] Jerry Mayfield: *Virtualization and Xen, an Open Source Approach*, `http://regions.cmg.org/regions/mspcmg/Presentations/May2006/-Virtualization%20and%20XEN.pdf`, 2006.

[4] John Scott Robin, Cynthia E. Irvine: *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*, Usenix Annual Technical Conference, `http://www.cs.nps.navy.mil/people/faculty/irvine/publications/-2000/VMM-usenix00-0611.pdf`, 2000.

[5] K. Lawton: *Running Multiple Operating Systems Concurrently on the IA32 PC Using Virtualization Techniques*, 1999.

[6] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, Larry Peterson: *Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors*, Princeton University, 2005.

[7] Poul-Henning Kamp, Robert N. M. Watson: *Jails: Confining the omnipotent root*, SANE 2000 proceedings, `http://phk.freebsd.dk/pubs/sane2000-jail.pdf` 2000.

[8] Intel Corporation: *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2004.

[9] Ed Silha, Cathy May, Brad Frey: *PowerPC Operating Environment Architecture, Book III*, IBM Corporation, 2003.

[10] Intel Corporation: *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, 2005.

[11] Andrew Whitaker, Marianne Shaw, Steven D. Gribble: *Denali: A Scalable Isolation Kernel*, Proceedings of the Tenth ACM SIGOPS European Workshop, 2002.

[12] The Xen Team: *Xen Interface Manual, Xen v3.0 for x86*, University of Cambridge, 2005.

[13] Fabrice Bellard: *QEMU, a Fast and Portable Dynamic Translator*, Usenix Annual Technical Conference, 2005.

Dodatek A

Konzole při bootu HelenOS/Xen

Systém byl nabootován v simulátoru Bochs.

```
 __  __            _____  ___  ____    ____
 \ \/ /___ _ __   |___ / / _ \|___ \  |___ \
  \  // _ \ '_ \    |_ \| | | | __) |__ __) |
  /  \  __/ | | |  ___) | |_| |/ __/|__/ __/
 /_/\_\___|_| |_| |____(_)___(_)_____| |_____|

 http://www.cl.cam.ac.uk/netos/xen
 University of Cambridge Computer Laboratory

 Xen version 3.0.2-2 (xenod@cl.cam.ac.uk) (gcc version 3.3.3
 20040412 (Red Hat Linux 3.3.3-7)) Thu Apr 13 17:34:06 BST 2006
 Latest ChangeSet: Thu Apr 13 15:18:37 2006 +0100 9617:5802713c159b

(XEN) Physical RAM map:
(XEN)  0000000000000000 - 000000000009fc00 (usable)
(XEN)  000000000009fc00 - 00000000000a0000 (reserved)
(XEN)  00000000000e8000 - 0000000000100000 (reserved)
(XEN)  0000000000100000 - 0000000008000000 (usable)
(XEN)  00000000fffc0000 - 0000000100000000 (reserved)
(XEN) System RAM: 127MB (130684kB)
(XEN) Xen heap: 10MB (10628kB)
(XEN) Using scheduler: Simple EDF Scheduler (sedf)
(XEN) PAE disabled.
(XEN) found SMP MP-table at 000fb0d0
(XEN) DMI not present.
(XEN) Using APIC driver default
(XEN) ACPI: Unable to locate RSDP
(XEN) Intel MultiProcessor Specification v1.4
(XEN)     Virtual Wire compatibility mode.
(XEN) OEM ID: BOCHSCPU Product ID: 0.1          APIC at: 0xFEE00000
(XEN) Processor #0 0:0 APIC version 20
(XEN) I/O APIC #1 Version 17 at 0xFEC00000.
(XEN) Enabling APIC mode:  Flat.  Using 1 I/O APICs
(XEN) Processors: 1
(XEN) Initializing CPU#0
```

```
(XEN) Detected 2.500 MHz processor.
(XEN) CPU0: AMD Flush Filter enabled
(XEN) CPU: L1 I~Cache: 0K (0 bytes/line), D cache 0K (0 bytes/line)
(XEN) Intel machine check architecture supported.
(XEN) Intel machine check reporting enabled on CPU#0.
(XEN) CPU0: AMD Athlon(tm) processor stepping 00
(XEN) Total of 1 processors activated.
(XEN) ENABLING IO-APIC IRQs
(XEN) ..TIMER: vector=0x31 apic1=0 pin1=0 apic2=-1 pin2=-1
(XEN) Platform timer is 1.193MHz PIT
(XEN) Brought up 1 CPUs
(XEN) Using IPI Shortcut mode
(XEN) *** LOADING DOMAIN 0 ***
(XEN) Domain 0 kernel supports features = { 00000001 }.
(XEN) Domain 0 kernel requires features = { 00000000 }.
(XEN) PHYSICAL MEMORY ARRANGEMENT:
(XEN)  Dom0 alloc.:   01c00000->02000000 (25676 pages to be
      allocated)
(XEN) VIRTUAL MEMORY ARRANGEMENT:
(XEN)  Loaded kernel: 80000000->800349a4
(XEN)  Init. ramdisk: 80035000->800489e9
(XEN)  Phys-Mach map: 80049000->80063130
(XEN)  Start info:    80064000->80065000
(XEN)  Page tables:   80065000->80067000
(XEN)  Boot stack:    80067000->80068000
(XEN)  TOTAL:         80000000->80400000
(XEN)  ENTRY ADDRESS: 800052a4
(XEN) Dom0 has maximum 1 VCPUs
(XEN) Initrd len 0x139e9, start at 0x80035000
(XEN) Scrubbing Free RAM: ..done.
(XEN) Xen trace buffers: disabled
(XEN) *** Serial input -> DOM0 (type 'CTRL-a' three times to switch
      input to Xen).
SPARTAN kernel, release 0.2.0.3 (Daylight), revision 1839
Built on 2006-08-01 04:30:37 for xen32
Copyright (C) 2001-2006 HelenOS project
kernel: 0x80000000 hardcoded_ktext_size=119K,
                   hardcoded_kdata_size=90K
stack:  0x80081000 size=4K
Xen memory: 0x67000 size: 108834816 (reserved 106496)
config.memory_size=104M
config.cpu_count=1
No init tasks found
cpu0: (AuthenticAMD family=15 model=2 stepping=0) 0MHz
```