



UNIWERSYTET ŁÓDZKI
WYDZIAŁ MATEMATYKI I INFORMATYKI

WIRTUALIZACJA SYSTEMÓW OPERACYJNYCH

Sławomir Wojciech Wojtczak

... znany także jako **vermaden** :)

118435/s

Łódź 2008

Praca wykonana pod kierunkiem
dr Mariusza Jarockiego
KATEDRA INFORMATYKI STOSOWANEJ

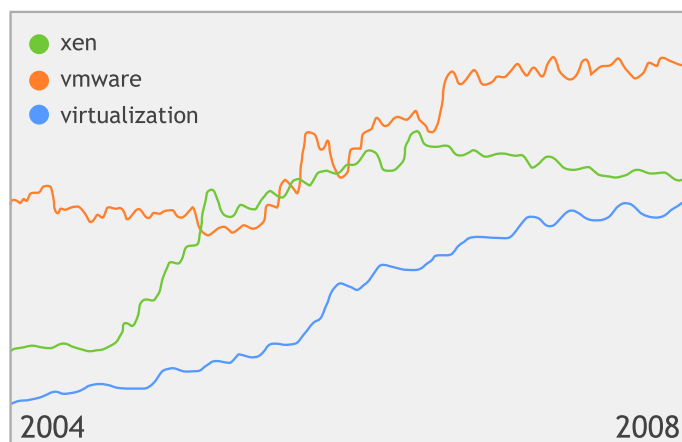
Spis treści

Wstęp	3
1 Wprowadzenie do wirtualizacji	5
1.1 Podstawowe pojęcia	5
1.2 Standardowe działanie systemu operacyjnego	6
1.3 Emulacja	8
1.4 Wirtualizacja procesora	9
1.4.1 Hypervisor typu 1	10
1.4.2 Hypervisor typu 2	12
1.4.3 Parawirtualizacja	13
1.4.4 Pełna wirtualizacja	15
1.4.5 Wirtualizacja na poziomie systemu operacyjnego	16
1.4.6 Hybrydowa wirtualizacja	17
1.4.7 Partycje sprzętowe	17
1.5 Wirtualizacja pamięci	18
1.5.1 Shadow Page Table	20
1.5.2 Nested Page Table	21
1.5.3 Memory Overcommitment	22
1.6 Wirtualizacja operacji I/O	23
1.7 Rozszerzenia sprzętowe wspierające wirtualizację	28
1.7.1 AMD	29
1.7.2 Intel	33
1.7.3 VIA	37
1.7.4 Podsumowanie	37
1.8 Innowacje i nowe technologie	38
1.8.1 Sterowniki parawirtualne	39
1.8.2 SMP w systemach guest	39
1.8.3 NUMA	40
1.8.4 Cache Coherent NUMA	41
1.8.5 Topology Aware Scheduler	42
1.8.6 Zarządzanie energią i skalowanie	44
1.8.7 Sprzętowa akceleracja grafiki	45
1.8.8 Seamless Mode	48
1.9 Zalety	50
1.10 Wady	52
1.11 Zastosowania	53

2	Dostępne maszyny wirtualne	55
2.1	Hypervisor typu 1	55
2.1.1	Xen	55
2.1.2	xVM	56
2.1.3	VMware ESX	57
2.2	Hypervisor typu 2	58
2.2.1	QEMU	58
2.2.2	VirtualBox	59
2.2.3	KVM	61
2.2.4	VMware Server	61
2.2.5	VMware Workstation	62
2.2.6	VMware Fusion	63
2.2.7	Parallels Desktop	63
2.2.8	Parallels Workstation	64
2.2.9	Parallels Server	64
2.2.10	Win4Lin / Win4BSD / Win4Solaris	65
2.3	Na poziomie systemu operacyjnego	66
2.3.1	FreeBSD Jails	66
2.3.2	Solaris Containers	67
2.3.3	Linux VServer	68
2.3.4	Linux OpenVZ	69
2.3.5	Parallels Virtuozzo Containers	70
2.3.6	User Mode Linux	71
2.3.7	Cooperative Linux	71
2.4	Emulatory	73
2.4.1	QEMU	73
2.4.2	Bochs	73
3	Wydajność maszyn wirtualnych	74
4	Przyszłość wirtualizacji	77
5	Podsumowanie	78
	Technikalia	79
	Bibliografia	79
	Spis rysunków	84
	Indeks	84

Wstęp

Wirtualizacja systemów operacyjnych to temat szeroki i skomplikowany. Praca ta ma na celu zebranie i dogłębne omówienie wszystkich aktualnie liczących się rozwiązań i nowych technologii w dziedzinie wirtualizacji. Omawia różne podejścia, nowe rozwiązania jak i te sprawdzone, używane od lat, dając pełny przekrój aktualnych możliwości w tej dziedzinie informatyki. Wirtualizacja pozwala na wydajne działanie "obcych" systemów operacyjnych na innym systemie, z większą bądź mniejszą integracją, a także na rozwiązania *stricte* przeznaczone pod współdziałanie setek czy tysięcy systemów operacyjnych. Omówione zostały wszystkie liczące się rozwiązania związane z wirtualizacją, zarówno darmowe jak i płatne z uwzględnieniem ich zalet, wad oraz ogólnego zastosowania.



Wzrost zainteresowania *wirtualizacją*, źródło danych **Google Trends**.

Już teraz wirtualizacja jest wykorzystywana na wielką skalę i to praktycznie wszędzie. Mnóstwo stron internetowych jest serwowanych na serwerach wirtualnych, właśnie dzięki wirtualizacji i rozwiązaniom opisanym w niniejszej pracy. Z czasem wirtualizacja stanie się jeszcze bardziej powszechna i niezauważalna zarazem, gdyż końcowy użytkownik w istocie wcale nie musi być świadom, że korzysta z jej dobrodziejstw. Stanie się także bardziej "przezroczysta", gdyż kolejne generacje procesorów wraz z rozszerzeniami następnej generacji jeszcze bardziej ułatwią jej stosowanie. Odpowiednio stosowana wirtualizacja pozwala na znaczne zmniejszenie kosztów stałych oraz tych związanych z infrastrukturą. Pozwala na znaczne oszczędności prądu, więc sprzyja ochronie środowiska.

Wirtualizacja to tak naprawdę przyszłość informatyki, są oczywiście pewne dziedziny, w których (jeszcze) nie ma ona tak wielkiego znaczenia czy też praktycznego zastosowania, jednak z czasem będzie się to zmieniało na korzyść wirtualizacji. Postanowiłem wybrać właśnie wirtualizację na temat mojej pracy magisterskiej, gdyż dziedzina ta dostarcza zupełnie nowych możliwości oraz innowacyjnych rozwiązań. Wirtualizacja zapewnia wiele ułatwień i udogodnień, zwłaszcza gdy korzystamy z więcej niż jednego systemu operacyjnego, co w sumie nie jest w dzisiejszych czasach aż taką rzadkością. Poza tym jest to aktualnie najprężniej rozwijająca się gałąź informatyki i najwięcej się tu dzieje.

Poniżej zamieszczam krótkie omówienie każdego z rozdziałów pracy.

Rozdział 1: Wprowadzenie do wirtualizacji

Podstawowe pojęcia niezbędne do zrozumienia działania wirtualizacji, metody jej realizowania, rozszerzenia sprzętowe wspierające wirtualizację. Rozdział ten zawiera także omówienie nowych technologii, a na koniec przedstawia wady, zalety i zastosowania wirtualizacji.

Rozdział 2: Dostępne maszyny wirtualne

Dogłębnie omawia dostępne na rynku rozwiązania. Zostały one podzielone na kategorie w celu ułatwienia porównywania oferowanych przez nie funkcjonalności. Dokładny opis używanych technik i rozwiązań oraz sugerowane zastosowania.

Rozdział 3: Wydajność maszyn wirtualnych

Mierzenie wydajności maszyn wirtualnych jest trudne, czasochłonne i skomplikowane, rozdział ten omawia najczęściej popełniane błędy przy próbie porównywania różnych rozwiązań wirtualizacji. Zawiera także sugestie i wskazówki jak należy porównywać maszyny wirtualne.

Rozdział 4: Przyszłość wirtualizacji

Aktualnie istnieje mnóstwo nowych rozwiązań w dziedzinie wirtualizacji, a w najbliższym czasie powstanie ich jeszcze więcej, rozdział ten po krótko omawia w jakich kierunkach będzie rozwijać się wirtualizacja, zarówno w krótszej jak i długiej perspektywie.

Rozdział 5: Podsumowanie

Podsumowanie zebranych informacji.

Dodatek **Technikalia** omawia szczegóły techniczne dotyczące samej pracy magisterskiej, ewentualnych problemów oraz narzędzi użytych przy jej powstawaniu. Mam nadzieję, że niniejsza praca przyczyni się do lepszego poznania i zrozumienia mechanizmów funkcjonowania wirtualizacji.

Rozdział 1

Wprowadzenie do wirtualizacji

1.1 Podstawowe pojęcia

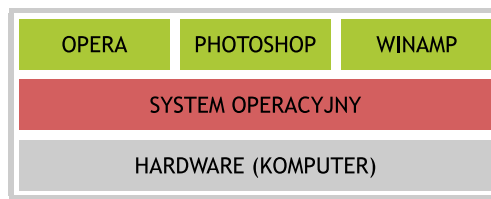
Wirtualizacja to bardzo ogólne pojęcie i aby dokładnie je wyjaśnić i zrozumieć należy najpierw poznać kilka podstawowych pojęć, dzięki którym będzie to możliwe. Jednocześnie zakładam, że czytający tą pracę zna podstawowe pojęcia i zagadnienia związane z tematyką komputerową jak *procesor* czy *pamięć* które można łatwo nadrobić w źródłach takich jak choćby [Wales, 2001]. Mimo wszystko postaram się wyjaśnić wszystkie zagadnienia w możliwie najłatwiejszy do zrozumienia sposób, zgodnie z ideą KISS¹.

Przez *host* będziemy rozumieć komputer z systemem operacyjnym, bądź też sam system operacyjny, na którym będziemy uruchamiać inne systemy operacyjne, przez *guest* zaś każdy z systemów operacyjnych uruchomionych na systemie *host*. W przypadku systemu *host* terminologia ta odnosi się też do sytuacji, w której system *host* ma rolę zarządzającą.

Z kolej *system operacyjny* to oprogramowanie, które zarządza sprzętem naszego komputera, czyli procesorami, pamięcią, dyskami twardymi, kartami graficznymi i wszystkim innym co znajduje się w tym pudle pod biurkiem w taki sposób, że możemy uruchamiać na tymże systemie najróżniejsze aplikacje, takie jak Firefox i Winamp, czy też grać w gry i pracować w najróżniejszych programach co przedstawia **rysunek 1.1**. Pojęcie komputer lub sprzęt będziemy zastępować również pojęciem *hardware*. Mówiąc bardziej technicznie system operacyjny zarządza pamięcią, procesami (czyli naszymi aplikacjami), plikami, połączeniami sieciowymi oraz wieloma innymi aspektami związanymi z codzienną pracą komputera.

Na rynku dostępnych jest bardzo wiele systemów operacyjnych, zarówno komercyjnych jak i systemów otwartych rozwijanych na wolnych licencjach pokroju *GPL* [Stallman, 1989], *BSD* [Computer Systems Research Group of the University of California, 1989] czy *CDDL* [Sun, 2004]. Nie będę się tutaj rozwodzić nad ich klasyfikacją i szufladkowaniem do odpowiednich kategorii, poza małymi wyjątkami są one ze sobą niekompatybilne, co oznacza, że jeżeli posi-

¹Keep It Small and Simple



Rysunek 1.1: Schemat działania systemu operacyjnego.

adamy program działający na jednym, to na drugim już go nie uruchomimy, chyba że skorzystamy z dobrodziejstwa jakie niesie wirtualizacja.

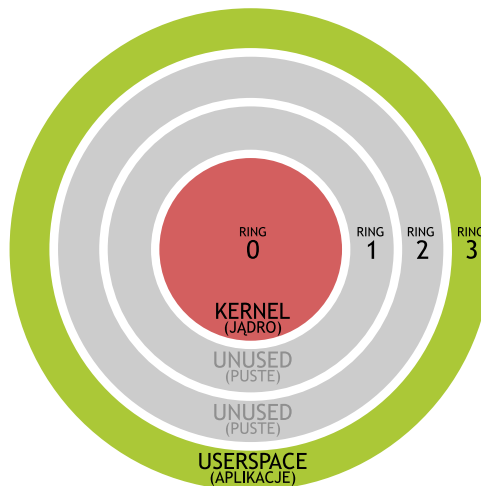
Mówimy tutaj oczywiście o kompatybilności binarnej owych aplikacji, istnieje oczywiście mnóstwo aplikacji specjalistycznych takich jak serwery www, bazy danych i mnóstwa innych aplikacji które zostały przeportowane na całą rodzinę systemów [Thompson, 1969], a często nawet na systemy Windows, ale trzeba je najpierw zbudować ze źródeł. Biorąc pierwszy lepszy program działający w systemie X, nie będzie on działał w systemie Y tak po prostu.

Wirtualizacja jest szczególnie przydatna, jeżeli korzystamy z systemu innego niż system z rodziny Windows. Producenci wielu programów komercyjnych takich jak Adobe Photoshop czy 3D Studio MAX nie wydają wersji na inne systemy, a mimo iż wolne oprogramowanie jak GIMP czy Blender dobrze sobie radzą jako zamienniki, to jednak czasami po prostu jesteśmy zmuszeni do produktów komercyjnych. Czy to przez naszego szefa lub firmę, czy po prostu przez własne przyzwyczajenia i wyuczoną już obsługę jednego produktu, nie będziemy przecież uczyć się od nowa całej aplikacji gdy mamy projekt do zrobienia na wczoraj.

1.2 Standardowe działanie systemu operacyjnego

Zobaczmy jak zachowuje się system operacyjny na sprzęcie natywnie. Każdy system ma do wykorzystania cztery *protection rings* (uprzywilejowane pierścienie). Numerowane są od 0 do 3. Kod, który działa w trybie *ring 0* ma wszystkie uprawnienia, może bezpośrednio "rozmawiać" ze sprzętem, w skrócie może robić ze sprzętem wszystko. Tryb *ring 3* przeznaczony jest zazwyczaj na tak zwany *userspace* (aplikacje), posiada najmniejsze uprawnienia i to tutaj właśnie działają aplikacje. System uprzywilejowanych pierścieni charakteryzuje hierarchiczna budowa, która jest mechanizmem ochronnym zapewniającym bezpieczeństwo, jest przedstawiona na **rysunku 1.2**.

Pozostałe dwa tryby, odpowiednio *ring 1* oraz *ring 2* są stosunkowo rzadko wykorzysty-



Rysunek 1.2: Typowe wykorzystanie *protection rings* przez system operacyjny.

wane, chociaż nic nie stoi na drodze programistów aby je wykorzystać. Głównym powodem takiego wyboru jest to, iż przełączanie pomiędzy trybami nie jest zbyt wydajne. Innym powodem takiego postępowania są również zależności historyczne związane z portowaniem kodu na inne architektury ponieważ wiele procesorów obsługuje tylko dwa tryby, uprzywilejowany i nieuprzywilejowany, czyli odpowiednio ring 0 oraz ring 3. Można również spotkać się z nazewnictwem *kernel mode* dla trybu uprzywilejowanego oraz *user mode* dla trybu nieuprzywilejowanego. Inna sprawa, że według wielu ludzi tryby pośrednie nie są wcale aż tak użyteczne.

Ogromna większość systemów wykorzystuje mechanizm pierścieni w następujący sposób, jądro systemu operacyjnego wraz ze sterownikami urządzeń działa w najważniejszym i najbardziej uprzywilejowanym trybie ring 0, aplikacje uruchomione na tym systemie działają w trybie ring 3, a pozostałe tryby "leżą odłogiem". W ten sposób napisana jest większość systemów z rodziny UNIX, oraz systemy z rodziny Windows.

Dobrym przykładem na wyjątek potwierdzający regułę jest tutaj system *OS/2*, który poza standardowymi trybami ring 0 oraz ring 3 używał również trybu ring 2 dla aplikacji z prawami dostępu do operacji *I/O* (wejścia/wyjścia).

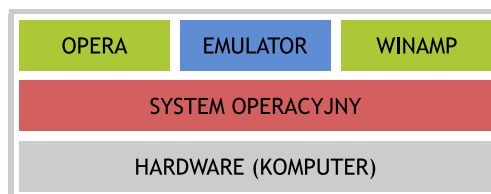
Systemy operacyjne, które wykorzystują *microkernel* (mikrojądro), również używają tylko dwóch trybów, gdzie małe jądro działa w trybie ring 0 (czasami mieści się nawet w pamięci L1 procesora), a serwery jądra oraz sterowniki działają w trybie ring 3 razem z aplikacjami. Przykładem takich systemów są na przykład *QNX* czy *MINIX 3*.

Mechanizm przechodzenia pomiędzy pierścieniami, zwany po prostu *gates* to specjalne bramki, które dają zewnętrznemu pierścieniowi dostęp do zasobów wewnętrznego pierścienia w pewien zdefiniowany sposób. Implementacja mechanizmu czy też po prostu systemu op-

eracyjnego, który w odpowiedni sposób wykorzysta bramki pomiędzy pierścieniami zapewni mu duże bezpieczeństwo, sprawiając, że zarówno źle napisane programy czy też sterowniki nie będą powodowały problemów na wysokości jądra. W razie problemów czy to program czy też sterownik byłby po prostu restartowany, a w razie problemów po prostu wyłączany. Dla przykładu przeglądarka internetowa nie mogłaby uzyskać dostępu do sieci bez odpowiedniego pozwolenia z zewnętrznego pierścienia. Niestety wadą takiego rozwiązania jest to, iż mechanizm przełączania pomiędzy pierścieniami nie jest zbyt wydajny i powoduje dodatkowe opóźnienia.

1.3 Emulacja

Na koniec wyjaśnijmy jeszcze czym jest *emulacja* systemu operacyjnego. Polega ona na stworzeniu wirtualnego środowiska, które w praktyce dla systemu guest jest postrzegane jako kompletny komputer. Wszystkie elementy wirtualnego komputera są programowo emulowane, między innymi takie jak CPU, RAM, HDD, GPU, BIOS i CD. Emulator, to więc nic innego jak kolejna aplikacja działająca w trybie ring 3, co przedstawia **rysunek 1.3**. Potrzebuje on jednak o wiele więcej zasobów niż typowa aplikacja aby realizować swoje zadania z sensowną szybkością.



Rysunek 1.3: *Emulator* jest po prostu kolejną aplikacją działającą w systemie.

Wielką zaletą emulacji jest możliwość emulowania systemów, które wymagają innej architektury sprzętowej niż architektura systemu host, na przykład emulacja architektury PowerPC na najpopularniejszej aktualnie architekturze i386, było to swego czasu popularne dzięki emulatorowi *PearPC*, który był wykorzystywany do uruchamiania systemu *Mac OS X* na systemach Windows. Aktualnie najpopularniejszymi aplikacjami, zapewniającymi emulację są na *QEMU*² oraz *Bochs*.

Niestety wielką wadą emulacji jest jej wydajność, a konkretnie jej brak. Guest jest często wielokrotnie wolniejszy od systemu host. Z biegiem czasu zaczęto szukać coraz to wydajniejszych sposobów na realizację emulacji dążąc do emulowania jak najmniejszej ilości za-

²QEMU wraz z modulem *kqemu* może również służyć jako maszyna wirtualna

sobów wirtualnego środowiska, a instrukcje gościa wykonywać bezpośrednio na systemie host. Z czasem pojawiło się jeszcze więcej rozwiązań mających na celu przyspieszenie działania systemu guest i tu właśnie tutaj kończy się emulacja a zaczyna wirtualizacja.

1.4 Wirtualizacja procesora

Wirtualizacja, podobnie jak emulacja, polega na stworzeniu na systemie host kompletnego wirtualnego środowiska dla systemu guest, jednak używając do tego specjalnych technologii lub też rozszerzeń sprzętowych i emulowaniu tylko tego czego nie da się zrealizować sprzętowo w celu zapewnienia jak największej wydajności systemu guest. W praktyce wydajność wirtualizacji jest porównywalna z wydajnością systemu host. Wadą takiego rozwiązania jest, że systemy host i guest muszą mieć tę samą architekturę sprzętową.

Zgodnie z tezą postawioną w "*Formal Requirements for Virtualizable Third Generation Architectures*" [Goldberg, 1974] czyli *Formalnych Wymagań wobec Wirtualizowalnych Architektur Trzeciej Generacji* wirtualizacja musi spełniać pewne warunki.

Maszyna wirtualna musi być zdolna do wirtualizacji całego sprzętu komputera, jego zasobów, łącznie z procesorami, pamięcią, przechowywaniem danych oraz peryferiów. Zaś *monitor maszyny wirtualnej* jest oprogramowaniem, które zapewnia abstrakcję maszynie wirtualnej. Zamiennym pojęciem dla monitora jest też *virtual machine monitor* a w skrócie *VMM*. Są trzy własności które według Popeka/Goldberga musi spełniać maszyna wirtualna:

- **Equivalence (równoważność)** - Program działający pod kontrolą wirtualizacji powinien zastać środowisko identyczne jak w przypadku bezpośredniego działania na tym sprzęcie.
- **Resource Control (kontrola zasobów)** - Monitor musi posiadać pełną kontrolę nad wirtualizowanymi zasobami.
- **Efficiency (wydajność)** - Instrukcje w większości muszą być wykonywane bez interwencji monitora.

Popek/Goldberg określają również charakterystykę *ISA - Instruction Set Architecture* (zestaw instrukcji) jakie fizyczna maszyna musi spełniać aby być w stanie realizować wymienione wyżej własności. Charakterystyka zawiera procesor działający w trybach *user mode* i *kernel mode* oraz ma dostęp do jednolitej liniowo adresowanej pamięci, operacje *I/O* oraz przerwania nie

są emulowane.

Twórca systemu operacyjnego OpenBSD czyli Theo de Raadt podzielił się swoimi przemyśleniami na temat bezpieczeństwa wirtualizacji na architekturze x86. Dotyczy to także separacji maszyn wirtualnych. Zgodnie z jego słowami, nie należy traktować wirtualizacji jako lepszy mechanizm bezpieczeństwa poprzez dokładanie kolejnych warstw, gdyż sama architektura x86 ze względu na swoją budowę i słaby mechanizm ochrony stron jest dosyć trudna w obsłudze, więc łatwo popełnić błędy przy projektowaniu zwykłego systemu operacyjnego, a co dopiero przy mechanizmie, który pozwoli uruchamiać kilka systemów operacyjnych na jednej maszynie. Poniżej zamieszczam owy cytat z listy dyskusyjnej.

"x86 virtualization is about basically placing another nearly full kernel, full of new bugs, on top of a nasty x86 architecture which barely has correct page protection. Then running your operating system on the other side of this brand new pile of shit. You are absolutely deluded, if not stupid, if you think that a worldwide collection of software engineers who can't write operating systems or applications without security holes, can then turn around and suddenly write virtualization layers without security holes. You've seen something on the shelf, and it has all sorts of pretty colours, and you've bought it."

Theo de Raadt

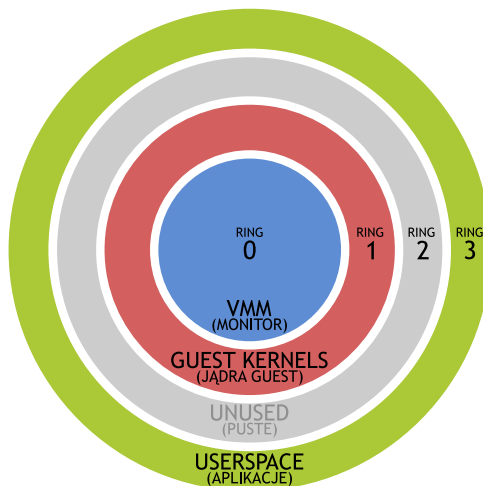
Hypervisor, zwany też często *virtual machine monitor* (monitor maszyny wirtualnej) to programowe narzędzie, które pozwala na jednoczesne działanie wielu systemów operacyjnych na jednym fizycznym komputerze. Każdemu z wirtualizowanych systemów "wydaje się" że mają bezpośredni dostęp do wszystkich zasobów komputera, chociaż tak naprawdę to monitor kontroluje i zarządza wszystkimi zasobami. Udostępnia on te zasoby, dzieląc je pomiędzy uruchomione systemy guest.

Możliwości wirtualizacji również są ograniczone do wykorzystania mechanizmu *protection rings*, więc w zależności od tego jak jest ona realizowana, wykorzystuje ona różne pierścienie owego mechanizmu. Jest to również zależne od wykorzystania do tego celu rozszerzeń sprzętowych komputera czy też nie.

1.4.1 Hypervisor typu 1

Hypervisor typu 1 zajmuje miejsce pomiędzy sprzętem komputera a systemami operacyjnymi guest, którym udostępnia zasoby sprzętowe komputera. Przedstawia go **rysunek 1.4**. Zazwyczaj jeden z systemów jest uprzywilejowany i służy jako domena zarządzająca pozostałymi

systemami guest. Sam monitor działa w trybie ring 0, podczas gdy systemy guest działają w przestrzeni ring 1, a aplikacje standardowo w trybie ring 3. Bez rozszerzeń sprzętowych rozwiązanie takie pozwala jedynie na *parawirtualizację*, o której więcej w dalszej części tego rozdziału.

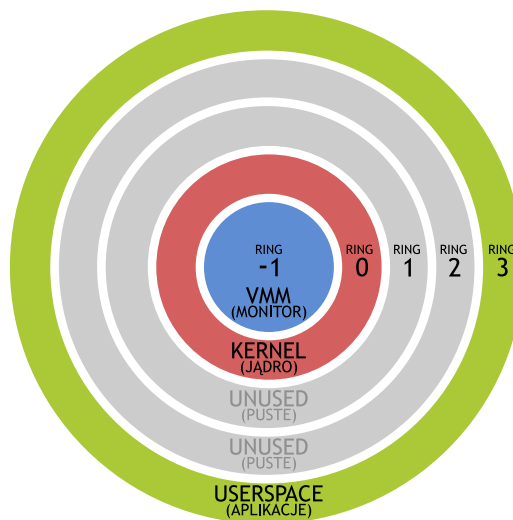


Rysunek 1.4: Schemat *hypervisora* typu 1 na procesorze bez obsługi *monitor mode*.

Jeżeli procesor posiada rozszerzenia sprzętowe wspierające wirtualizację, to schemat wykorzystania pierścieni nieco się zmienia. Rozszerzenia sprzętowe dostarczają również dodatkowe tryby, *root mode* oraz *non root mode*. Każdy z tych trybów zawiera niezależne tryby ring 0-3, *hypervisor* typu 1 działa w trybie *root mode*, mając pełny dostęp do prawdziwego sprzętu, podczas gdy niezmodyfikowany guest działa w standardowych dla siebie trybach ring 0-3 w trybie *non root mode*, a jego dostęp do sprzętu jest w pełni kontrolowany przez hypervisora.

Wirtualizacja wspierana sprzętowo oferuje instrukcje, które wspierają bezpośrednie odwołania parawirtualizowanego gościa do hypervisora. W praktyce możemy to logicznie opisać w następujący sposób, niezmodyfikowane systemy guest działają w standardowych dla siebie przestrzeniach, czyli ring 0 dla jądra oraz ring 3 dla *userspace*, a host działa w trybie *ring -1* (minus jeden) będąc wyżej w hierarchii pierścieni. Tryb ten zwany jest też zamiennie *VMX root*. Schemat takiego działania przedstawia **rysunek 1.5**.

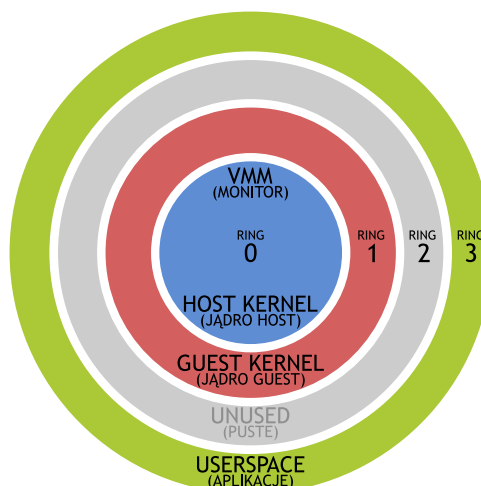
Tryb który zapewniają rozszerzenia sprzętowe zwany jest również *monitor mode*. Guest tak naprawdę nie wie, że ma do dyspozycji jedynie ograniczone kontrolowane przez hypervisora zasoby. Dzięki temu hypervisor nie musi wykonywać wielu skoków pomiędzy trybami ring 1 i ring 0 co znacznie zwiększa wydajność wirtualizacji. Inną zaletą jest to, że dzięki owym rozszerzeniom możliwe jest uruchamianie guestów bez ich modyfikacji, więc możliwa jest *pełna wirtualizacja* systemu guest.



Rysunek 1.5: Schemat *hypervisora typu 1* na procesorze z obsługą *monitor mode*.

1.4.2 Hypervisor typu 2

Hypervisor typu 2 zwany również *hosted hypervisor* działa pod kontrolą systemu operacyjnego jak każda inna aplikacja. W tym wypadku to system operacyjny pełni rolę hosta, mając pod kontrolą cały sprzęt i udostępnia go jedynie hypervisorowi. Hypervisor część instrukcji wykonuje bezpośrednio na procesorze hosta, jak kod każdej innej aplikacji, musi jednak mieć dostęp do przestrzeni jądra systemu hosta, co przeważnie realizowane jest po prostu modulem jądra dla hypervisora. Emulowane są tylko te elementy maszyn wirtualnych, których nie da się zrealizować sprzętowo, ani też bezpośrednio na systemie hosta. Schemat tego typu hypervisora przedstawia **rysunek 1.6**.



Rysunek 1.6: Schemat *hypervisora typu 2* na mechanizmie *protection rings*.

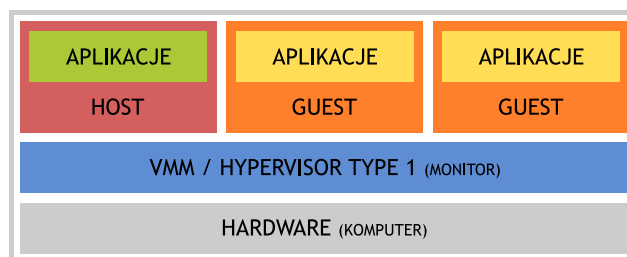
Jedną z głównych metod realizowania wirtualizacji przez *hypervisor typu 2* jest *binary translation*, czyli tłumaczenie instrukcji gościa w locie na instrukcje, które będą wykonane bezpośred-

nio na procesorze za pomocą systemu host, z kolej bezpośrednie wykonywanie instrukcji gościa na systemie host jak kod każdej innej aplikacji zwane jest *direct execution* (bezpośrednie wykonywanie). Translacja odbywa się pomiędzy instrukcjami jądra gościa, które rezyduje w trybie ring 1, a jądrem systemu host pracującym w trybie ring 0. Aplikacje standardowo urzędują w trybie ring 3, zarówno hosta jak i gościa. Wiele hypervisorów w implementuje również mechanizm pamięci podręcznej dla zapytań gościa, czyli tak zwane *cache*, co jeszcze bardziej usprawnia działanie gościa. Inną również często stosowaną techniką jest kompilacja *just in time*³, instrukcja jest kompilowana podczas jej pierwszego wywołania, a każde następne jej wywołanie powoduje jedynie ponowne uruchomienie już skompilowanej wersji.

Przyjęło się uznawać, że *hypervisor typu 2* realizuje wirtualizację jedynie przez mechanizmy jak *binary translation* czy *direct execution*. Istnieją już jednak rozwiązania jak *KVM*, które całą swoją funkcjonalność opierają na wykorzystaniu sprzętowych rozszerzeń wirtualizacji czyli *AMD-V* oraz *VT-x*, co więcej bez owych rozszerzeń nie będą w ogóle działać.

1.4.3 Parawirtualizacja

Słowo *para* pochodzi z języka greckiego i znaczy dosłownie obok, a *parawirtualizacja* to nic innego jak jedna lub wiele maszyn wirtualnych działających obok systemu host. Parawirtualizacja wykorzystuje hypervisor typu 1, monitor działa bezpośrednio na sprzęcie udostępniając go (lub nie) systemom guest. Jeden z wirtualizowanych systemów jest uprzywilejowany i pełni rolę zarządzającą, może rozdzielać zasoby pomiędzy pozostałe systemy guest i ma pełną kontrolę nad maszyną wirtualną i jej zasobami. Schemat działania *parawirtualizacji* przedstawia rysunek 1.7.



Rysunek 1.7: Parawirtualizacja oraz pełna wirtualizacja przy użyciu hypervisora typu 1.

Parawirtualizacja nie różni się aż tak bardzo od techniki *binary translation*. Mechanizm *binary translation* tłumaczy odpowiednie instrukcje systemu guest w locie na odwołania hyper-

³Zwana też czasami jako *dynamic recompilation*.

visor (inaczej *hypercall*) podczas gdy parawirtualizacja robi to samo, tylko na poziomie kodu źródłowego. Oczywiście modyfikacja kodu daje o wiele więcej możliwości, między innymi eliminację niepotrzebnych odwołań systemu guest. Poza tym translacja w locie musi odbywać się szybko, więc nie może posiadać zbyt kompleksowej implementacji gdyż nie będzie zapewniała odpowiedniej wydajności.

Aby system mógł być uruchomiony jako guest za pomocą parawirtualizacji, trzeba zmodyfikować jego jądro tak, aby zamiast odwołań do sprzętu czy też standardowych odwołań tego systemu, które nie są dozwolone podczas parawirtualizacji, używał odwołań *hypercall* bezpośrednio do hypervisora. Dotyczy to odwołań do pamięci, obsługi przerwań, operacji I/O i wielu innych rzeczy. Odwołanie *hypercall* to nic innego jak wywołanie systemowe systemu guest, z żądaniem pewnych zasobów lub też zmian od hypervisora. Idea jest taka sama jak przy zwykłym odwołaniu systemowym *syscall* do jądra systemu operacyjnego. Odwołania *hypercall* to po prostu interfejs pomiędzy monitorem a systemami guest. Mechanizm ten zapewnia bardzo dobrą wydajność, zbliżoną do natywnego działania systemu.

Zaletą parawirtualizacji jest bardzo dobra wydajność systemów guest oraz to, iż stosunkowo łatwo jest zmodyfikować jądro systemu guest aby przystosować go parawirtualizacji, nie trzeba też używać mechanizmu *binary translation* który jest dosyć trudny w implementacji, a jego prostsze implementacje nie zapewniają tak dużej wydajności. Jej wadą jest właśnie owa konieczność modyfikacji systemu guest, ponieważ nie zawsze jest to możliwe. Systemy, których źródła nie są dostępne nie mogą być zmodyfikowane. Dodatkowo modyfikacja jądra niesie za sobą pewne koszty związane z dostosowaniem systemu guest do danego hypervisora i musimy tą procedurę powtórzyć dla każdego innego hypervisora.

VMI (Virtual Machine Interface)

Implementacja parawirtualizacji w postaci VMI (czyli *Virtual Machine Interface*) to interfejs komunikacji pomiędzy systemem guest a hypervisorem zaproponowany przez VMware, z czasem specyfikacja została otwarta. Można powiedzieć, że VMI to po prostu próba ujednolicenia interfejsu dla wirtualizacji. Celem tego posunięcia było także stworzenie jednego interfejsu komunikacji, który mógłby być zaimplementowany na wielu hypervisorach i byłby w miarę łatwy do implementacji na systemach guest. Specyfikacja VMI mówi o następujących celach:

- **przenośność** - interfejs powinien być łatwy do zaadoptowania w systemach guest.
- **wydajność** - implementacja musi zapewniać dobrą wydajność.
- **konserwacja** - upgrade i zarządzanie systemem guest powinno być proste i łatwe.
- **rozszerzalność** - API powinno być skonstruowane w taki sposób aby można było je w przyszłości rozszerzyć w prosty sposób.

Aktualnie interfejs VMI działa z jądrem Linux 2.6, w przyszłości VMware ma nadzieję na oficjalną adopcję/implementację ich interfejsu przez wszystkie liczące się systemy z rodziny UNIX, takie jak BSD, Solaris, Darwin. Zaletą VMI jest fakt, że implementacja ta nie jest przypisana do tylko jednego hypervisora oraz to, że jest otwarta. Implementacja systemu guest raz dałaby możliwość uruchomienia go we wszystkich liczących się hypervisorach.

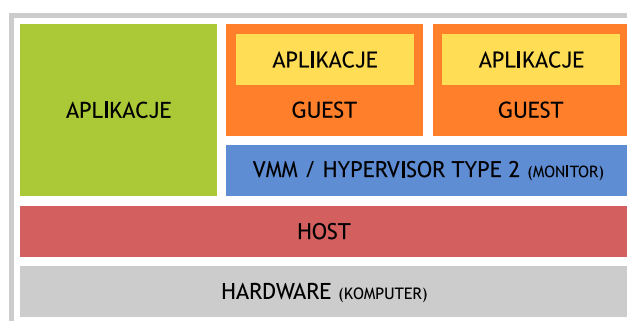
1.4.4 Pełna wirtualizacja

Pełna wirtualizacja pozwala na wirtualizowanie dowolnego niezmodyfikowanego systemu operacyjnego. Jest to z pewnością największa zaleta pełnej wirtualizacji ponieważ nie narzuca nam żadnych ograniczeń ani wymagań co do systemu guest. Wydajność takiego rozwiązania niestety jest nieco mniejsza niż w przypadku parawirtualizacji. Pełna wirtualizacja może być realizowana zarówno za pomocą hypervisora typu 1 jak i typu 2.

Głównym wyzwaniem dla pełnej wirtualizacji jest przechwytywanie i realizowanie uprzywilejowanych instrukcji jądra guesta, jak na przykład operacji I/O, przerwań czy operacji na pamięci. Większość instrukcji systemu guest może być wykonana bezpośrednio na procesorze przez hypervisora, ale te instrukcje, które wychodzą poza uprawnienia systemu guest, czy też odwołują się do sprzętu, muszą być przechwycone i emulowane.

W przypadku użycia hypervisora typu 1 wymagane są odpowiednie rozszerzenia sprzętowe umożliwiające wirtualizację, takie jak *AMD-V* czy *Intel VT-x*, zostaną one opisane dokładniej w jednym z kolejnych podrozdziałów. Schemat działania jest tutaj taki sam jak w przypadku *parawirtualizacji*, co przedstawia **rysunek 1.7**. Pełna wirtualizacja określana jest też zamiennie pojęciami *natywna wirtualizacja* oraz *wirtualizacja wspierana sprzętowo*.

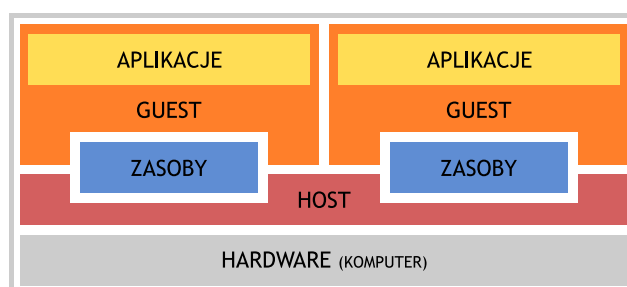
Jeżeli zaś do pełnej wirtualizacji użyjemy hypervisora typu 2, to będzie ona realizowana przez mechanizmy takie jak *binary translation* oraz *direct execution*, często ze wsparciem pamięci podręcznej owych translacji w postaci *cache*, można też użyć mechanizmu kompilacji instrukcji *just in time*. Drugi przypadek prezentuje **rysunek 1.8**.



Rysunek 1.8: Pełna wirtualizacja realizowana za pomocą hypervisora typu 2.

1.4.5 Wirtualizacja na poziomie systemu operacyjnego

Wirtualizacja na poziomie systemu operacyjnego czyli *OS level virtualization* w przeciwieństwie do poprzednich metod nie używa hypervisora. System guest jest zawsze dokładnie taki sam jak system host, co więcej zarówno host jak i wszystkie wirtualizowane systemy guest korzystają bezpośrednio z jądra systemu host. Możemy nawet powiedzieć, że używają tego samego jądra, jednak nie wszystkie z takim samym dostępem jak system host. Systemy guest, tak jak w każdym innym modelu wirtualizacji są oczywiście odseparowane zarówno od siebie jak i od systemu host. Każdy z wirtualizowanych systemów guest jest jedynie replikacją struktury systemu host, mniej lub bardziej zmodyfikowaną. Schemat tej metody przedstawia rysunek 1.9.



Rysunek 1.9: Schemat wirtualizacji na poziomie systemu.

Podjęcie to posiada wiele unikalnych zalet w porównaniu do poprzednich rozwiązań. W przypadku klasycznej wirtualizacji każdy z systemów guest przy użyciu hypervisora komunikuje się ze sprzętem przez specjalną warstwę abstrakcji powodując dodatkowy narzut i opóźnienie. Sprawia to, że im więcej maszyn wirtualnych mamy uruchomionych, tym więcej zasobów jest zużywanych jedynie na samą warstwę abstrakcji i komunikację ze sprzętem. W przypadku wirtualizacji na poziomie systemu operacyjnego wszystkie te problemy nie istnieją ponieważ każdy z systemów guest korzysta bezpośrednio z zasobów komputera w taki sam sposób jak system host. Oczywiście dzieje się to za przyzwoleniem systemu host. Rozwiązanie to powoduje, że systemy guest są praktycznie identycznie wydajne jak system host.

Mówiąc inaczej jądro systemu host pozwala na uruchomienie wielu niezależnych instancji *userspace*. Istnieje wiele konwencji nazw dla tego typu wirtualizacji, niektóre z nich to *containers* czyli tak zwane *kontenery*, można spotkać się też z określeniem *virtual environment* czyli *wirtualne środowisko*. Mechanizm *wirtualizacji na poziomie systemu* jest logicznym i funkcjonalnym rozwinięciem funkcjonalności *chroot* z systemów UNIX, z resztą to właśnie głównie na tych systemach tego typu wirtualizacja występuje. Współczesne implementacje tego typu wirtualizacji zapewniają oczywiście zarządzanie zasobami systemów guest, ustawianie priorytetu dla każdego z systemów oraz limity czasu procesora i pamięci.

1.4.6 Hybrydowa wirtualizacja

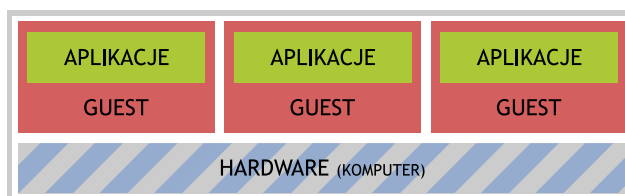
Zarówno *pełna wirtualizacja* jak i *parawirtualizacja* posiadają swoje wady i zalety. Pełna wirtualizacja pozwala na uruchamianie praktycznie każdego systemu operacyjnego bez konieczności jego modyfikacji jednak cierpi na tym wydajność. Parawirtualizacja z kolei zapewnia bardzo dobrą wydajność zbliżoną do natywnej, jednak wymaga odpowiedniego przystosowania systemu operacyjnego do odwołań hypervisora. Technika która stara się wykorzystać zalety wyżej wymienionych technik przy okazji omijając ich wady jest *hybrydowa wirtualizacja*.

Technika ta polega na tym, że system guest działa na zasadzie pełnej wirtualizacji, jednak posiada zainstalowane odpowiednie parawirtualizowane sterowniki do urządzeń. Sprawia to, że dostęp do sprzętu nie cierpi z powodu dodatkowego narzutu. Podejście takie nie wymaga również modyfikacji systemu guest oraz zapewnia bardzo dobrą wydajność. Parawirtualizowane sterowniki używane są do operacji I/O oraz obsługi przerw.

1.4.7 Partycje sprzętowe

Całkiem inne podejście w stosunku do już poznanych oferują *partycje sprzętowe* zwane też *hardware partitions*. Rozwiązanie to polega na podzieleniu zasobów sprzętowych na partycje. Na każdej takiej partycji możemy uruchomić dowolny niezmodyfikowany system operacyjny. Każda z partycji zachowuje się jak kompletny komputer z procesorem, pamięcią, operacjami I/O czy siecią. Niektórzy producenci sprzętu do tego typu wirtualizacji jak HP oferują nawet rozwiązania pozwalające na dostarczenie osobnego źródła zasilania dla każdej partycji. Partycje sprzętowe są też zamiennie nazywane pojęciami *server entity* czy też *logical domain*. Partycje

sprzętowe realizowane są przez *firmware* sprzętu, czyli niskopoziomowy program działający bezpośrednio na sprzęcie, jak BIOS płyty głównej na przykład. Podział zasobów pomiędzy określone partycje odbywa się za pomocą specjalnej konsoli, którą możemy też uruchomić przez sieć. Schemat partycji sprzętowych przedstawia **rysunek 1.10**.



Rysunek 1.10: Schemat działania rozwiązania *hardware partitions*.

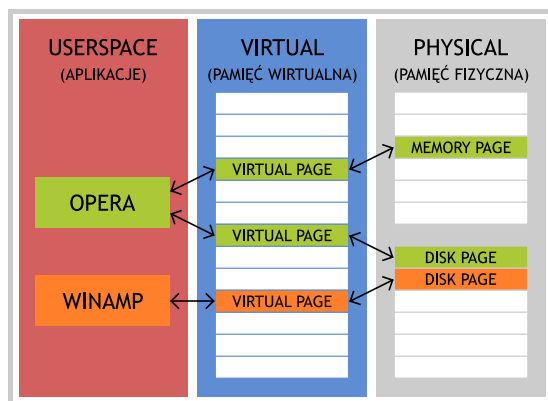
Ogólnie uważa się, że *partycje sprzętowe* będą używane coraz rzadziej i powoli odejdą w zapomnienie. Najprawdopodobniej staną się po prostu drogim wyspecjalizowanym rozwiązaniem dla wąskiego grona odbiorców, jak na przykład banki i wojsko. Rozwiązanie takie posiada też pewne ograniczenia, na przykład ilość uruchomionych systemów w tym samym czasie jest ograniczona do minimum jednego procesora na system. Jeżeli mamy więc serwer z 32 procesorami to będziemy mogli uruchomić w najlepszym przypadku 32 systemy operacyjne. Istnieją rozwiązania pozwalające na dzielenie procesorów pomiędzy partycje, jednak stanowią one mniejszość.

Rozwiązanie takie jest bardzo nieefektywne. Serwery przez większą część czasu mimo wszystko pozostają bezczynne, w przypadku wirtualizacji ma to o tyle znaczenie, że podczas gdy niektóre systemy "nudzą się" inne wykorzystują wszystkie przydzielone im zasoby. Partycje sprzętowe nie pozwalają aby wiele systemów współdzieliło zasoby tego samego procesora a co za tym idzie, wiele zasobów i prądu zostanie zmarnowane, a przecież jedną z celów wirtualizacji jest *konsolidacja* serwerów. Do zalet partycji sprzętowych można i trzeba zaliczyć jednak w pewnym sensie pewność takiego rozwiązania. Żadne oprogramowanie nie jest wolne od błędów, a *bug* w hypervisorze może zawiesić działanie wszystkich maszyn wirtualnych. Każda z partycji jest zupełnie niezależna od pozostałych partycji, jednak *firmware* to również swego rodzaju oprogramowanie i tam też mogą wkrąść się błędy.

1.5 Wirtualizacja pamięci

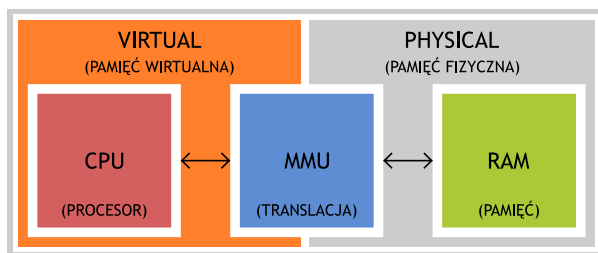
Poza wirtualizacją procesora kluczowa jest także odpowiednia *wirtualizacja pamięci*. We współczesnych systemach operacyjnych najczęstszym rozwiązaniem w dziedzinie zarządza-

nia pamięci jest *virtual memory* czyli *pamięć wirtualna*. Jedynym wyjątkiem od tej reguły są niektóre systemy czasu rzeczywistego *RTOS* czyli *real time operating system* dla których największe znaczenie mają jak najmniejsze opóźnienia. *Pamięć wirtualna* to technika, która daje aplikacjom złudzenie że mają dostęp do zunifikowanej przestrzeni adresowej, chociaż "pod maską" może być to kilka oddzielnych modułów pamięci o różnych rozmiarach oraz plik wymiany na dysku twardym. Schemat *pamięci wirtualnej* przedstawia **rysunek 1.11**.



Rysunek 1.11: Schemat działania *pamięci wirtualnej*.

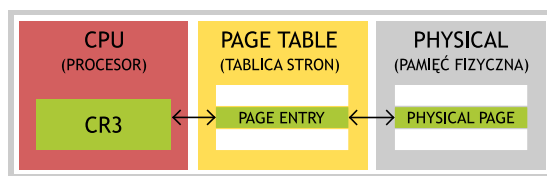
W sytuacji gdy na komputerze działa jeden system operacyjny fizyczne adresy pamięci są tłumaczone na adresy wirtualne (oraz na odwrót) przez układ *memory management unit* (w skrócie *MMU*). Pamięć jest dzielona na strony (inaczej *pages*) o pewnym ustalonym rozmiarze i w ten sposób przydzielana odpowiednim aplikacjom przez system operacyjny. Schemat działania *MMU* przedstawia **rysunek 1.12**.



Rysunek 1.12: Schemat działania układu *memory management unit*.

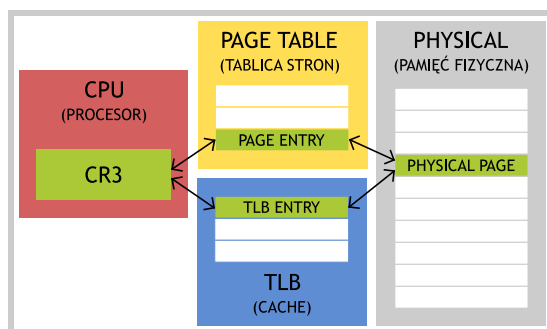
Dawniej, w starszych generacjach procesorów *stronicowanie* (czyli *paging*) odbywało się przeważnie jedynie za pośrednictwem *page table* (inaczej *tablica stron*). System przechowuje informacje o wolnych i używanych ramkach w tablicy stron, która służy do translacji adresów logicznych na fizyczne. Schemat działania tego rozwiązania przedstawia **rysunek 1.13**.

Aby przyspieszyć ową translację stosuje się dodatkowe rozwiązanie zwane *translation lookaside buffer* (w skrócie *TLB*). Bufor ten służy jako pamięć podręczna *cache* w postaci tablicy asocjacyjnej dla ostatnio używanych stron pamięci. Układ *TLB* przyspiesza cały proces



Rysunek 1.13: Schemat translacji przy użyciu *page table*.

translacji ponieważ od razu otrzymujemy adres potrzebnej strony w pamięci fizycznej, dodatkowo tłumaczenie adresów odbywa się niezależnie od standardowej procedury translacji. Jeżeli żadanego adresu nie ma w buforze TLB, wtedy adres fizyczny jest używany do odnalezienia poszukiwanych danych w pamięci. Mówiąc ogólnie aktualny adres strony jest przechowywany w rejestrze CR3 procesora czyli tak zwanym *hardware page table pointer* a najczęściej używane strony przechowywane w buforze TLB. Schemat działania bufora TLB przedstawia **rysunek 1.14**. W dzisiejszych czasach każdy nowy procesor posiada układ TLB.



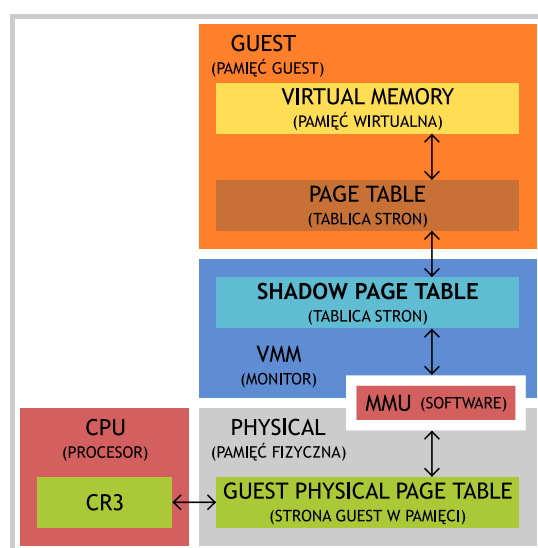
Rysunek 1.14: Schemat translacji przy użyciu bufora TLB.

Wiemy już w jaki sposób pamięć jest używana w przypadku gdy na komputerze system operacyjny działa samodzielnie. Czas poznać na jakie sposoby jest ona wykorzystywana w przypadku wirtualizacji. System guest oczekuje standardowej pamięci adresowanej od zera, jak w przypadku natywnego działania. Zarządzanie wirtualną pamięcią dla systemów guest nie różni się generalnie od zarządzania pamięcią wirtualną w dzisiejszych systemach operacyjnych. Jest to po prostu ciągła przestrzeń adresowa, która jest logicznie powiązana z modułami pamięci zamontowanymi w komputerze. System operacyjny przechowuje mapowanie wirtualnych numerów stron do stron fizycznych w specjalnych tablicach. Maszyna wirtualna zaś jest odpowiedzialna za mapowanie pamięci systemów guest na fizyczną pamięć komputera.

1.5.1 Shadow Page Table

W przypadku wirtualizacji fizyczna pamięć dzielona jest pomiędzy poszczególne systemy guest oraz system host. Sam hypervisor również alokuje pamięć na własny użytek oraz rozdziela ją pomiędzy systemy guest i host. Występuje tu jednak pewien problem natury technicznej,

problem dwustopniowej translacji adresów w przypadku wirtualizacji. System guest otrzymuje od hypervisora pewien obszar pamięci i zarządza nim wedle własnego uznania. Z drugiej strony nie jest to obszar przydzielony bezpośrednio w pamięci, lecz jedynie w pamięci hypervisora, który to z kolei przydziela tę pamięć z pamięci fizycznej komputera. Zachodzi więc problem podwójnego kopiowania obszarów pamięci co jest niestety nie po drodze z wydajnością. Rozwiązanie to nosi nazwę *shadow page table*. Jest to nic więcej jak emulowanie jednostki MMU przez hypervisor. W takim rozwiązaniu systemy guest otrzymują wirtualny emulowany rejestr CR3 przez który mają dostęp do pamięci. Technikę *shadow page table* przedstawia **rysunek 1.15**.

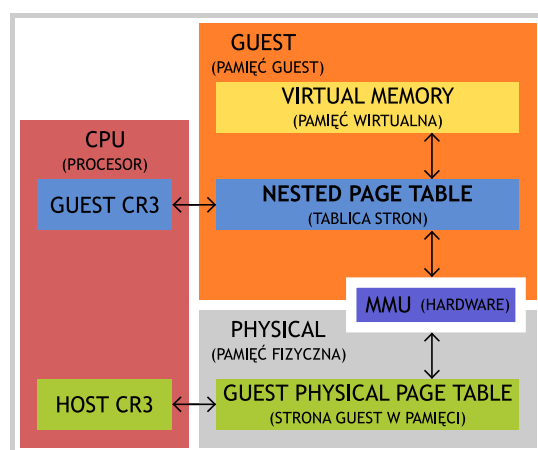


Rysunek 1.15: Schemat mechanizmu *shadow page table*.

1.5.2 Nested Page Table

Rozwiązaniem tego problemu jest sprzętowa wirtualizacja jednostki MMU w buforze TLB, rozwiązanie to nosi nazwę *nested page table*. System host posiada swój rejestr CR3 czyli *host CR3* a systemom guest udostępniany jest rejestr *guest CR3*. Przez mapowanie pamięci systemy guest mogą przydzielać pamięć bezpośrednio w pamięci komputera i nie cierpi na tym wydajność. Taki bufor TLB posiada specyficzną flagę zwaną *Address Space Identifier*, która wskazuje, do którego systemu guest należy dany obszar pamięci. Rozwiązanie to nazywane jest też zamiennie jako *Tagged TLB*. Inną zaletą tego rozwiązania jest to, że im więcej systemów guest działa na raz, tym więcej zyskujemy na wydajności. Procesor musi po prostu zsynchronizować bufor TLB tak samo jakby działało się to natywnie bez wirtualizacji. Analogiczne, wirtualizacja przy braku owej wirtualizacji w buforze TLB działa wolniej z każdym kolejnym wirtualizowanym systemem guest. Technikę *nested page table* przedstawia **rysunek 1.16**.

Sprzętowa wirtualizacja jednostki MMU w buforze TLB mimo wszystkich dobrodziejstw jakie ze sobą niesie posiada również jedną wadę. Rozwiązanie *nested page table* komplikuje



Rysunek 1.16: Schemat mechanizmu *nested page table*.

całą procedurę standardowej translacji adresu jeżeli nie ma go w tablicy. Musimy wykonać czterokrotnie więcej kroków podczas takiej translacji niż w przypadku braku sprzętowej wirtualizacji jednostki MMU. Istnieje jednak bardzo proste rozwiązanie tego problemu, współczesne bufory TLB są bardzo duże, więc sytuacja ze standardową translacją ma miejsce bardzo rzadko.

1.5.3 Memory Overcommitment

Przy wirtualizacji pamięci nie można nie wspomnieć o mechanizmach *memory overcommitment*, czyli przydzielania większej ilości pamięci systemom guest niż jest jej dostępnej fizycznie w systemie. Definicja ta może początkowo brzmieć niedorzecznie, wyjaśnijmy więc na czym owe rozwiązanie polega. Przydzielamy systemom guest więcej pamięci niż jest fizycznie dostępnej, jednak większość z nich (jeżeli nie wszystkie nawet) nie wykorzystują całej przydzielonej im pamięci, a jedynie jej część, pamięć z której systemy guest aktualnie nie korzystają może być wykorzystana na inne systemy guest, a gdy będzie potrzebna danemu systemowi, będzie wtedy "zabrana" pozostałym. Można to w skrócie ująć jako dynamiczne przydzielanie pamięci, w zależności od potrzeb, ale dla każdego system mimo to definiujemy maksymalną ilość możliwej przydzielonej pamięci.

Nie jest to rozwiązanie uniwersalne, nie zawsze można, czy też opłaca się z niego skorzystać. Na przykład w środowiskach gdzie mamy do czynienia z systemami guest, o których wiemy, że będą wykorzystywać większość przydzielonej im pamięci. Innym przykładem środowiska nie nadającego się na stosowanie tego mechanizmu, będą systemy guest, które często przydzielają i zwalniają duże ilości pamięci, w tym przypadku mechanizm będzie działał, jednak ze względów wydajności stosowanie go tutaj nie będzie najlepszym wyjściem. Narzut związany z *memory overcommitment* nie jest jakiś strasznie duży, ale jest to rozwiązanie stworzone dla środowisk, w których pracuje wiele systemów guest, które przez większość czasu nie zajmują prawie całej swojej przydzielonej pamięci.

Mechanizm *memory overcommitment* może być realizowany na kilka sposobów, każdy z nich nadaje się do innego rodzaju środowiska.

Memory Ballooning

Rozwiązanie *memory ballooning* pozwala systemom guest dynamicznie zmieniać rozmiar przydzielonej pamięci, po prostu systemy guest mają zwracać nieużywaną pamięć przez specjalny mechanizm (przeważnie moduł lub usługa jądra dla systemu guest). Wadą tego rozwiązania jest poleganie na "dobrej woli" systemu guest przy zwalnianiu pamięci, co niestety nieco zmniejsza tak zwane *reliability* tego rozwiązania.

Transparent Page Sharing

W sytuacji, gdy nasze środowisko składa się wielu podobnych czy też identycznych systemów guest (czy chociażby z identycznymi aplikacjami) o wiele lepszym rozwiązaniem jest mechanizm *transparent page sharing* (zwany też czasami *content based page sharing*). Identyczne strony pamięci są łączone w jedną, im więcej identycznych stron pamięci, tym więcej pamięci oszczędzamy. W razie zapisu jednego z systemów guest stosowany jest mechanizm *copy on write* aby stworzyć nową stronę o innej zawartości. Wadą tego rozwiązania jest z pewnością skomplikowana implementacja oraz spory spadek wydajności w środowiskach które nie są przez większą część czasu w stanie *idle*. Nie wymaga jednak modułu jądra czy też usługi po stronie systemu guest, jest więc to mechanizm przezroczysty.

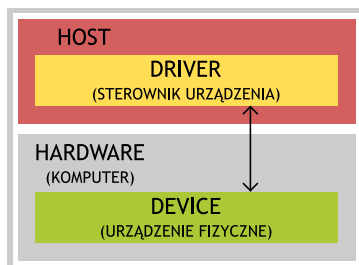
Swapping

Ostatnim z mechanizmów *memory overcommitment* jest *swapping*, podobnie jak poprzednie rozwiązanie nie wymaga "uświadamiania" systemu guest o jego istnieniu. Polega na implementacji jednej przestrzeni *swap* dzielonej w razie potrzeby przez wszystkie systemy guest. Niestety posiada ono wiele wad. Przez przezroczystość hypervisor wie mniej o aktualnym wykorzystaniu pamięci przez poszczególne systemy guest. Dodatkowo istnieje dosyć spora szansa, że hypervisor będzie dublował mechanizm *swapowania* systemów guest, inną kwestią jest skomplikowana implementacja takiego rozwiązania.

1.6 Wirtualizacja operacji I/O

Trzecia i ostatnia z kluczowych warstw wirtualizacji to *wirtualizacja operacji I/O*. Bez wirtualizacji system operacyjny odwołuje się po prostu bezpośrednio do urządzeń komputera przez

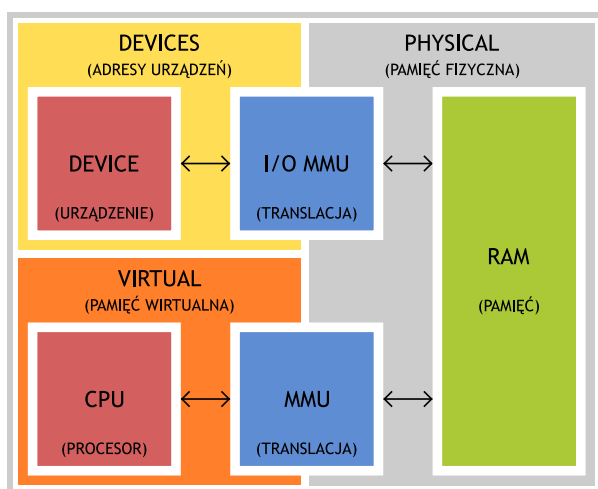
odpowiedni sterownik, udostępniając (lub nie) dane urządzenie aplikacjom przez interfejs sterownika. Przedstawia to **rysunek 1.17**.



Rysunek 1.17: Schemat dostępu do urządzenia *natywnie*.

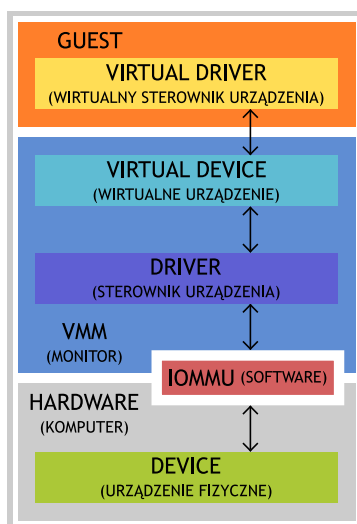
Dawniej kiedy kopiowanie danych z urządzeniami odbywało się programowo przez tryb *PIO*, procesor był przez cały czas zajęty transferowaniem danych i nie mógł w tym czasie robić żadnych innych rzeczy. Nie muszę chyba pisać jak bardzo było to nieefektywne. Z czasem wprowadzono mechanizm *DMA*, który pozwala na przesyłanie danych znaczenie mniejszym kosztem i mniejszym obciążeniem procesora niż tryb *PIO*. Umożliwia on urządzeniom bezpośredni zapis i odczyt z fizycznej pamięci niezależnie od procesora. W praktyce wygląda to tak, że CPU rozpoczyna transfer i wraca do innych czynności podczas gdy transfer odbywa się w tle, potem odbiera jedynie przerwania od kontrolera *DMA*, że operacja się zakończyła.

Tak samo jak układ *MMU* tłumaczy adresy stron logicznych na fizyczne, tak samo układ *I/O MMU* (w skrócie *IOMMU*) tłumaczy fizyczne adresy urządzeń na ich logiczne odpowiedniki w pamięci. W przypadku układu *IOMMU* czynność ta nazywa się *DMA remapping* czyli *mapowanie DMA*. Często odpowiada także za mapowanie przerwania w podobny sposób jak dzieje się to dla adresów stron oraz za kontrolę dostępu do danego urządzenia. Zestawienie układów *MMU* oraz *IOMMU* przedstawia **rysunek 1.18**.



Rysunek 1.18: Porównanie działania układów *MMU* oraz *I/O MMU*.

Innym od dawna stosowanym układem pokroju IOMMU jest układ *GART* stosowany przy mapowaniu pamięci kart graficznych. Układ IOMMU robi dokładnie to samo, tylko dla wszystkich urządzeń w komputerze, nie tylko dla karty graficznej. Urządzenia korzystające z kanału DMA swoje działania opierają na adresach fizycznych a nie logicznych, stąd też hypervisor nie ma możliwości zawężenia zakresu pamięci w przypadku transferu danych przy użyciu mechanizmu DMA do zakresu pamięci konkretnego systemu guest. W rezultacie hypervisor musi emulować stare, proste w budowie, dobrze znane urządzenia, gdyż są one łatwe do zaimplementowania, ponieważ nie posiadają skomplikowanych funkcjonalności współczesnych urządzeń. Oczywiście rozwiązanie takie, mimo bardzo dużej elastyczności przy zarządzaniu, bardzo cierpi na wydajności. Rozwiązanie takie nosi także nazwę programowego układu IOMMU. Opisany powyżej sposób emulowania urządzeń przez hypervisor przedstawia **rysunek 1.19**.

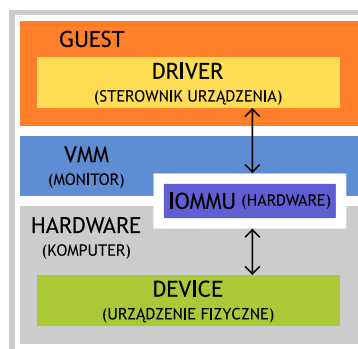


Rysunek 1.19: Programowe emulowanie układu *IOMMU* przez *hypervisor*.

Rozwiązaniem tego problemu jest właśnie wspomniany sprzętowy układ IOMMU. Pozwala on tłumaczyć fizyczne strony pamięci systemu guest, na adresy fizyczne w pamięci komputera, a dzięki temu hypervisor może pozwolić systemowi guest na bezpośrednią kontrolę nad fizycznym urządzeniem. Rozwiązanie jest to o tyle uniwersalne, że system guest może używać "zwykłych" niewirtualizowanych sterowników do komunikacji z tymże urządzeniem. Rozwiązanie to pozwala zmniejszyć niepożądane opóźnienia wynikające z virtualizacji do minimum.

Rozwiązanie to posiada niestety pewną wadę, jeżeli transfer danych nie powiedzie się kontroler DMA nie zwróci błędu, nie pozostaje zatem nic innego jak obsłużenie tego wyjątku programowo przez hypervisor. Mimo tego mankamentu rozwiązanie to jest zdecydowanie lepszym podejściem niż rozwiązanie czysto programowe i jest dobrym krokiem w stronę kolejnych bardziej rozwiniętych układów wspierających virtualizację. Sprzętowy układ I/O MMU

przedstawia **rysunek 1.20**.



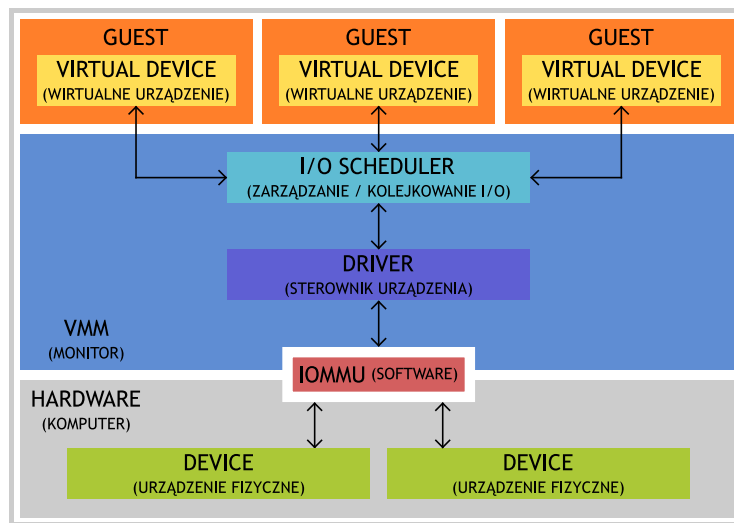
Rysunek 1.20: Schemat sprzętowego układu *IOMMU*.

Pozostają jeszcze inne problemy do rozwiązania związane z wirtualizacją operacji I/O. Mianowicie współdzielenie jednego, bądź kilku fizycznych urządzeń przez kilka systemów guest oraz/lub system host. Nie można w końcu pozwolić na dowolne korzystanie ze sprzętu ponieważ wtedy tak naprawdę żaden z systemów niczego nie osiągnie, bo jeden będzie kasował to co poprzedni dopiero ustawił i vice versa. W końcu to nie pamięć, że jeden system będzie korzystał ze swojej części a drugi ze swojej. Należy także pamiętać, że owe dzielenie się odnosi się nie tylko do tak podstawowych zagadnień jak karty sieciowe czy karty dźwiękowe, ale także karty graficzne i generowanie na nich akcelерованego obrazu 2D/3D, czy też skomplikowanych obliczeń wykonywanych bezpośrednio na GPU (zwane także *GPGPU*).

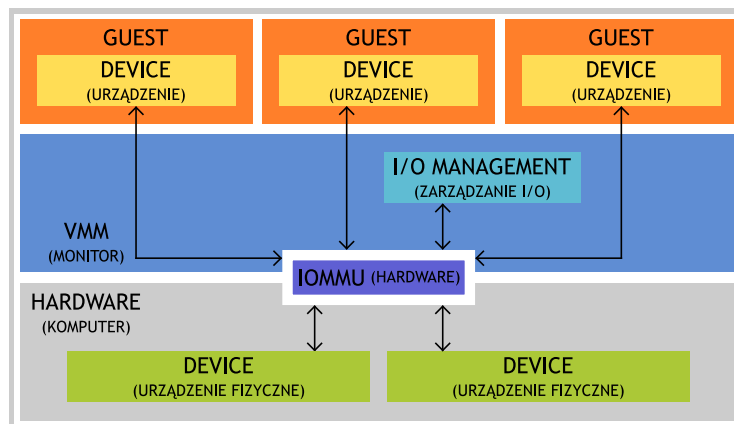
Podobnie jak w przypadku bezpośredniego dostępu do urządzeń również tutaj możemy użyć podejścia programowego jak i sprzętowego. W tym pierwszym wypadku hypervisor będzie odpowiedzialny za rozdzielanie, kolejkovanie i zarządzanie żadaniami I/O systemów guest. Będzie też musiał emulować urządzenia dla systemów guest i dopiero "osobiście" zajmować się wymianą danych z fizycznym sprzętem, a potem z powrotem rozdzielać na wirtualne urządzenia. Oczywiście dalej pozostaje problem podwójnego kopiowania pomiędzy urządzeniami wirtualnymi a fizycznymi. Daje to większe możliwości zarządzania i kontroli, jednak cierpi na tym wydajność, czasami dość znacznie. Programowe dzielenie urządzeń obrazuje **rysunek 1.21**.

Lepszym rozwiązaniem, na pewno w kwestii wydajności, jest z pewnością sprzętowe kolejkovanie i zarządzanie, znacznie odciąża procesor, a ponadto omija problem podwójnego kopiowania jak w przypadku programowego podejścia. Sprzętowe podejście do dzielenia urządzeń obrazuje **rysunek 1.22**.

Kolejną istotną kwestią jest całkowite "przekazywanie" fizycznych urządzeń konkretnym systemom guest w trybie wyłącznym (*dedicated*). Rozwiązanie to jest o tyle ważne i użyteczne, gdyż system host może nie posiadać odpowiednich sterowników do danego urządzenia, a sys-



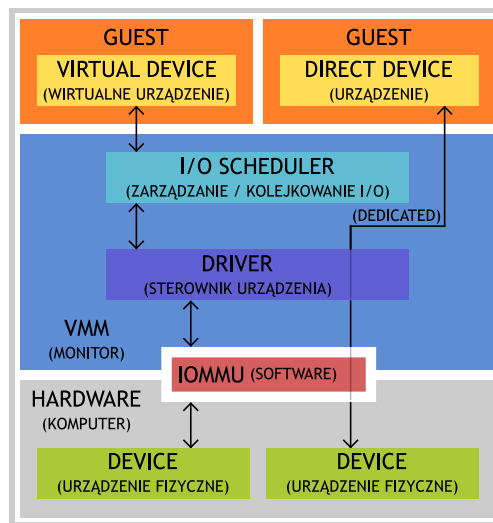
Rysunek 1.21: Schemat dzielenia urządzeń za pomocą programowych urządzeń wirtualnych.



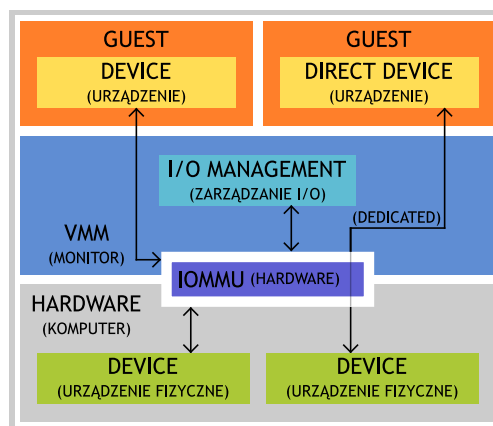
Rysunek 1.22: Schemat dzielenia urządzeń za pomocą rozwiązania sprzętowego.

tem guest jak najbardziej. Powyższa funkcjonalność jest użyteczna zarówno w środowisku serwerowym jak i desktopowym. W przypadku serwerów będzie to na pewno obsługa wszelkiej maści kontrolerów czy też oprogramowania zarządzającego do nich. Podobnie w przypadku całej masy urządzeń sieciowych (*switch/router*). W przypadku desktop przychodzą na myśl wszelkiej maści urządzenia USB jak kamery internetowe, odtwarzacze, aparaty cyfrowe. Przekazywanie urządzeń podobnie jak w przypadku ich dzielenia również może być zrealizowane programowo i sprzętowo. Programowe rozwiązanie przedstawia **rysunek 1.23**.

Porównując podejścia programowe i sprzętowe w przypadku "przekazywania" dochodzimy do tych samych wniosków jak w przypadku dzielenia urządzeń. Większa (często znacznie) wydajność rozwiązania sprzętowego oraz dużo mniejsze obciążenie procesora(ów). W przypadku dzielenia urządzeń pewnym argumentem dla rozwiązania programowego była większa elastyczność oraz możliwości, w tym wypadku rozwiązanie programowe nie ma tej zalety, bo w sumie nie ma i czym zarządzać, system guest dostaje pod swoją kontrolę urządzenie i robi z nim co chce. Podejście sprzętowe przedstawia **rysunek 1.24**.



Rysunek 1.23: Schemat programowego "przekazywania" urządzeń do systemu guest.



Rysunek 1.24: Schemat dzielenia urządzeń za pomocą rozwiązania sprzętowego.

Mimo pewnych zalet podejścia programowego jak większe możliwości zarządzania to podejście sprzętowe jest tym właściwszym i nie należy go rozpatrywać jako alternatywę a jako emulację w stosunku do podejścia programowego.

1.7 Rozszerzenia sprzętowe wspierające wirtualizację

Opisane wcześniej techniki wirtualizacji korzystają ze specjalnych sprzętowych rozszerzeń, zostaną one opisane w tym rozdziale.

1.7.1 AMD



Rysunek 1.25: Logo procesorów firmy *Advanced Micro Devices*.

Nazwa kodowa sprzętowych rozszerzeń wirtualizacji w procesorach AMD to *Pacifica*.

AMD-V / SVM

Rozszerzenie procesorów AMD *AMD-V* zwane też często *SVM* pozwala na sprzętową wirtualizację dostarczając specjalne tryby *root mode* (dla hypervisora) oraz *non root mode* (dla systemów guest) co pozwala na natywną wirtualizację systemów guest w trybie ring 0, a nie jak dotychczas w ring 1. Eliminuje także potrzebę programowego przechwytywania czułych instrukcji systemów guest.

Odpowiednik w procesorach Intel: **VT-x**

AMD Device Exclusion Vector

Device Exclusion Vector w skrócie *DEV* pozwala na kontrolę dostępu do pamięci systemów guest, co zwiększa bezpieczeństwo wirtualizacji i zapewnia izolację każdego z systemów guest. Rozwiązanie takie jest możliwe jedynie na procesorach ze zintegrowanym kontrolerem pamięci, patrz rozszerzenie AMD *Integrated Memory Controller*.

Odpowiednik w procesorach Intel: **(brak)**

AMD Integrated Memory Controller

Wirtualizacja to zadanie bardzo pamięciochłonne. Zintegrowany kontroler pamięci czyli *Integrated Memory Controller* (w skrócie *IMC* w procesorze zapewnia o wiele większą wydajność pamięci w porównaniu do historycznego rozwiązania bazującego na szynie *FSB*. Pamięć jest podłączona bezpośrednio do procesora co pozwala na często znaczne zmniejszenie opóźnień w dostępie do pamięci. Rozwiązanie takie świetnie sprawdza się też w konfiguracjach wieloprocessorowych (czyli *SMP* gdyż każdy procesor korzysta z podobnej szyby w dostępie do pamięci a nie jak w przypadku *FSB* dzieli ją z wszystkimi innymi procesorami. Architektura

taka określana jest też mianem *NUMA*.

Odpowiednik w procesorach Intel: **Integrated Memory Controller**

AMD Direct Connect Architecture

Rozwiązanie *Direct Connect Architecture* pozwala wyeliminować wąskie gardła tradycyjnego podejścia w komunikacji pomiędzy procesorem a urządzeniami opartego na szynie *FSB*. Rozwiązanie *DCA* pozwala połączyć procesor, kontroler pamięci i urządzenia magistralą *HyperTransport*. Pozwala to znacząco zwiększyć skalowalność i wydajność w komunikacji z urządzeniami.

Odpowiednik w procesorach Intel: **VT-c**

AMD Extended Migration

Mechanizm *Extended Migration* umożliwia migracje maszyny wirtualnej z jednego procesora (lub kilku) AMD na inny (inne) bez przerywania pracy systemu guest. Pozwala także na odpowiednie "ukrycie" różnic pomiędzy różnymi generacjami procesorów AMD pozwalając na migracje nawet pomiędzy różnymi generacjami procesorów.

Odpowiednik w procesorach Intel: **FlexMigration**

AMD IOMMU

Sprzętowa wirtualizacja operacji I/O czyli *IOMMU* (co rozwija się na *I/O Memory Management Unit*). Hypervisor będzie mógł dzielić urządzenia pomiędzy systemy guest bezpośrednio bez emulacji i podwójnego kopiowania, tłumaczenie adresów DMA, adresów urządzeń oraz sprawdzanie odpowiednich uprawnień dostępu do owego sprzętu.

Odpowiednik w procesorach Intel: **VT-d / VT-c**

AMD Rapid Virtualization Indexing

Implementacja mechanizmu *nested page table* w procesorach AMD nosi nazwę *Rapid Virtualization Indexing*.

Odpowiednik w procesorach Intel: **Extended Page Table**

AMD Address Space Identifier

Rozwiązanie *Address Space Identifier* w skrócie *ASID* to implementacja rozwiązania *Tagged TLB* w procesorach AMD pozwala na przypisanie obszarów pamięci konkretnym systemom guest, dzięki czemu nie trzeba za każdym razem opróżniać bufora TLB gdy procesor zmienia kontekst pomiędzy hypervisorem a systemem guest, znacznie zmniejsza to ilość operacji na pamięci potrzebnych do wykonania.

Odpowiednik w procesorach Intel: **Virtual Processor Identifier**

AMD HyperTransport

Technologia *HyperTransport* to szyna systemowa wymyślona w miejsce starszego brata o nazwie *FSB*. Zapewnia wysoki transfer danych, niskie opóźnienia oraz jest rozwiązaniem typu *point-to-point* mającym na celu przyspieszenie komunikacji pomiędzy urządzeniami. Jest kompatybilna z istniejącymi rozwiązaniami a jednocześnie może być rozszerzona do *Systems Network Architecture*. Jest przezroczysta dla systemu operacyjnego. Logo konsorcjum *HyperTransport* przedstawia rysunek 1.26.



Rysunek 1.26: Logo *HyperTransport Consortium*.

Odpowiednik w procesorach Intel: **QuickPath Interconnect**

AMD Cool'n'Quiet

Rozwiązanie *Cool'n'Quiet* to nic więcej jak mechanizm zarządzający napięciem i częstotliwością procesora w zależności od zapotrzebowania na moc obliczeniową. Jeżeli procesor nic nie robi (lub robi bardzo mało), napięcie jest obniżane razem z częstotliwością procesora. Pozwala to oszczędzać energię oraz wydzielać znacznie mniej ciepła, pozwala to także na cichsze chłodzenie. Stąd też wywodzi się nazwa rozwiązania, *Cool'n'Quiet* znaczy ni mniej ni więcej jak chłodno i cicho. Rozwiązanie to jest desktopową i serwerową wersją rozwiązania *PowerNow!*, które jest używane w Laptopach i innych urządzeniach mobilnych.

Rozwiązanie to występuje w serwerowych procesorach *Opteron* serii 'E' ale tam nosi nazwę *Optimized Power Management*, gdzie występuje w nieco zmodyfikowanej formie aby móc obsłużyć rejestrowaną pamięć RAM.

Odpowiednik w procesorach Intel: **(Enhanced) SpeedStep**

AMD (Enhanced) AMD PowerNow!

Technologia *PowerNow!* to podobnie jak w przypadku *Cool'n'Quiet* skalowanie prędkości procesora i jego napięcia, tyle że przeznaczona do urządzeń mobilnych.

Odpowiednik w procesorach Intel: **(Enhanced) SpeedStep**

AMD Enhanced Power Management

W skład technologii pod nazwą *Enhanced Power Management* wchodzi kilka mniejszych rozwiązań, związanych z oszczędzaniem energii i skalowaniem procesorów, są to:

Independent Dynamic Core

Pozwala regulować szybkość każdego z rdzeni procesora.

Dual Dynamic Power Management

Pozwala zarządzać prędkością i napięciem każdego procesora z osobna w systemach SMP.

CoolCore

Sprawdza czy procesor, pamięć, czy też obydwa są aktualnie potrzebne.

W razie potrzeby zmniejsza napięcie dla nieużywanych tranzystorów.

Odpowiednik w procesorach Intel: **(Enhanced) SpeedStep**

AMD Presidio

Rozszerzenie *Presidio* dostarcza mechanizmów bezpieczeństwa dla aplikacji. Każda aplikacja uruchamiana jest w swoim odizolowanym od pozostałych aplikacji środowisku. Niестety użycie tych rozszerzeń wymaga modyfikacji systemu operacyjnego.

Rozszerzenie współdziała z modulem *Trusted Platform Module* który zaprojektowała grupa *Trusted Computing Group*⁴.

Ma to na celu zabezpieczenie pamięci karty graficznej (podmiana obrazu przez inną aplikację), urządzeń typu *input* (program może sprawdzać co pisze użytkownik), pamięci RAM (czytanie pamięci innego programu) oraz kontrolera DMA (również kontrola pamięci). Za te

⁴Więcej informacji o grupie *Trusted Computing Group* na stronie <http://trustedcomputinggroup.org/>



Rysunek 1.27: Logo grupy *Trusted Computing Group*.

zabezpieczenia odpowiadają następujące technologie:

Isolated Execution Space (CPU)
Enhanced Virus Protection (CPU)
Storage Sealing (Trusted Platform Modules)
Secure Initialization (Chipset/CPU/Trusted Platform Modules)
Secure Input (Chipset/CPU)
Secure Output (Chipset/CPU/GPU)
Remote Attestation (Chipset/CPU/Trusted Platform Modules)

Odpowiednik w procesorach Intel: **Trusted Execution Technology**

Dostępność wyżej wymienionych rozszerzeń w procesorach AMD

http://en.wikipedia.org/wiki/List_of_AMD_Turion_microprocessors
http://en.wikipedia.org/wiki/List_of_AMD_Sempron_microprocessors
http://en.wikipedia.org/wiki/List_of_AMD_Athlon_64_microprocessors
http://en.wikipedia.org/wiki/List_of_AMD_Athlon_X2_microprocessors
http://en.wikipedia.org/wiki/List_of_AMD_Phenom_microprocessors
http://en.wikipedia.org/wiki/List_of_AMD_Opteron_microprocessors
http://en.wikipedia.org/wiki/Athlon_64
http://en.wikipedia.org/wiki/AMD_K10
<http://en.wikipedia.org/wiki/Opteron>

1.7.2 Intel

Nazwa kodowa sprzętowych rozszerzeń wirtualizacji w procesorach Intel nosi nazwę *Vanderpool*.



Rysunek 1.28: Logo procesorów firmy Intel.

Intel VT-x / VT-i

Rozszerzenia procesorów Intel VT-x (dla procesorów architektury i386/amd64) oraz VT-i (dla procesorów Itanium) pozwalają na sprzętową wirtualizację dostarczając specjalne tryby *root mode* (dla hypervisora) oraz *non root mode* (dla systemów guest) co pozwala na natywną wirtualizację systemów guest w trybie ring 0, a nie jak dotychczas w ring 1. Eliminuje także potrzebę programowego przechwytywania czułych instrukcji systemów guest.

Odpowiednik w procesorach AMD: **AMD-V / SVM**

Intel VT-d

Technologia VT-d zwana też marketingowo *Virtualization Technology for Directed I/O* to sprzętowa wirtualizacja operacji I/O spod znaku Intela. Dzięki niej hypervisor będzie mógł dzielić urządzenia pomiędzy systemy guest bez emulacji i podwójnego kopiowania, pozwala także na tłumaczenie adresów DMA, adresów urządzeń oraz sprawdzanie odpowiednich uprawnień dostępu do sprzętu.

Odpowiednik w procesorach AMD: **IOMMU**

Intel VT-c

Rozwiązanie VT-c nazywane zamiennie *Virtualization Technology for Connectivity* ma na celu wyeliminowanie wąskich gardeł w komunikacji pomiędzy procesorem a urządzeniami. Pozwala to na zwiększenie skalowalności i wydajności komunikacji z urządzeniami. Zawiera kilka mniejszych technologii:

Virtual Machine Device Queues

Zwane także *VMDq*, zwiększa przepustowość I/O dla rozwiązań sieciowych.

I/O Acceleration Technology

Zamiennie *I/OAT* minimalizuje zjawisko wąskiego gardła dla kart sieciowych.

Single-Root Input/Output Virtualization

Czyli *SR-IOV*, pozwala hypervisorowi na bezpośredni dostęp do sprzętu sieciowego.

Pozwala partycjonować szynę PCI/PCI Express na wirtualne interfejsy.

Odpowiednik w procesorach AMD: **Direct Connect Architecture / IOMMU**

Intel Extended Page Tables

Implementacja mechanizmu *nested page table* w procesorach Intel, nazwana *Extended Page Tables*.

Odpowiednik w procesorach AMD: **Rapid Virtualization Indexing**

Intel (Enhanced) Speedstep

W skrócie nazywane *EIST*, jest rozwiązaniem pozwalającym na skalowanie szybkości procesora i obniżania jego napięcia w zależności od zapotrzebowania na moc obliczeniową. Ma to na celu oszczędzanie energii oraz mniejsze wydzielanie ciepła podczas pracy w trybie *idle*.

Odpowiednik w procesorach AMD: **Cool'n'Quiet / (Enhanced) AMD PowerNow! / Enhanced Power Management**

Intel QuickPath Interconnect

Technologia *QuickPath Interconnect* to szyna systemowa mająca na celu zastąpić strasznie wysłużonej w rozwiązaniach Intela szynę *FSB*. Podobnie jak w przypadku rozwiązania AMD jest rozwiązaniem typu *point-to-point*, zapewnia wysoki transfer danych, niskie opóźnienia,

Odpowiednik w procesorach AMD: **HyperTransport**

Intel FlexPriority

Rozwiązanie *FlexPriority* udostępnia systemom guest dostęp do wirtualnego rejestru procesora sterującego priorytetami *task priority register* co odciąża hypervisor od emulowania i przechwytywania instrukcji systemów guest związanych z tymże rejestrem.

Odpowiednik w procesorach AMD: **AMD-V / SVM**

Intel FlexMigration

Technologia *FlexMigration* pozwala na migrację maszyny wirtualnej bez przerywania jej pracy, czyli tak zwaną *live migration* pomiędzy różnymi generacjami procesorów firmy Intel.

Odpowiednik w procesorach AMD: **Extended Migration**

Intel Virtual Processor Identifier

Rozwiązanie *Virtual Processor Identifier* w skrócie *VPID* to implementacja rozwiązania *Tagged TLB* w procesorach Intel. Pozwala na przypisanie obszarów pamięci konkretnym systemom guest, dzięki czemu nie trzeba za każdym razem opróżniać bufora TLB gdy procesor zmienia kontekst pomiędzy hypervisorem a systemem guest, znacznie zmniejsza ilość operacji na pamięci potrzebnych do wykonania.

Odpowiednik w procesorach AMD: **Address Space Identifier**

Intel Trusted Execution Technology

Rozszerzenia o nazwie *Trusted Execution Technology* zwane w skrócie *TXT* zapewniają bezpieczeństwo działania aplikacji. Każda aplikacja uruchamiana jest w swoim odizolowanym od pozostałych aplikacji środowisku. Nazwa kodowa tego rozwiązania to *LaGrande*. Rozszerzenie współdziała z modułem *Trusted Platform Module* grupy *Trusted Computing Group*.

Technologia ta ma na celu zabezpieczenie pamięci karty graficznej (podmiana obrazu przez inną aplikację), urządzeń typu *input* (program może sprawdzać co pisze użytkownik), pamięci RAM (czytanie pamięci innego programu) oraz kontrolera DMA (również kontrola pamięci).

Odpowiednik w procesorach AMD: **Presidio**

Dostępność wyżej wymienionych rozszerzeń w procesorach Intel

<http://processorfinder.intel.com>

http://en.wikipedia.org/wiki/List_of_Intel_Atom_microprocessors

http://en.wikipedia.org/wiki/List_of_Intel_Pentium_4_microprocessors

http://en.wikipedia.org/wiki/List_of_Intel_Pentium_D_microprocessors

http://en.wikipedia.org/wiki/List_of_Intel_Core_microprocessors

http://en.wikipedia.org/wiki/List_of_Intel_Core_2_microprocessors
http://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors
http://en.wikipedia.org/wiki/List_of_Intel_Itanium_microprocessors
[http://en.wikipedia.org/wiki/Intel_Core_\(microarchitecture\)](http://en.wikipedia.org/wiki/Intel_Core_(microarchitecture))
[http://en.wikipedia.org/wiki/Intel_Nehalem_\(microarchitecture\)](http://en.wikipedia.org/wiki/Intel_Nehalem_(microarchitecture))
http://en.wikipedia.org/wiki/Intel_Core_i7

1.7.3 VIA

Procesory spod znaku VIA również obsługują sprzętową wirtualizację, obsługują ją najnowsze procesory VIA NANO o kodowej nazwie *Isaiah*.



Rysunek 1.29: Logo procesorów firmy VIA.

Posiadają implementację zgodną z rozszerzeniami procesorów Intel, czyli VT-i.

Dostępność wyżej wymienionych rozszerzeń w procesorach VIA

http://en.wikipedia.org/wiki/List_of_VIA_Nano_microprocessors
http://en.wikipedia.org/wiki/VIA_Nano

1.7.4 Podsumowanie

Po lekturze tegoż podrozdziału można odnieść wrażenie, że zarówno AMD jak i Intel oferują bardzo podobne, praktycznie takie same rozwiązania, jedynie pod innymi nazwami. Prawda jest taka, że wielu z opisanych tutaj rozszerzeń firmy Intel nie ma w aktualnie dostępnych procesorach z serii *Core 2* czy też *Xeon*, zostały jednak opisane ze względu na bardzo bliską premierę procesora *Core i7* o kodowej nazwie *Nehalem*, który wnosi naprawdę dużo w kwestii wirtualizacji. Z drugiej strony wszystkie zastosowane w nim rozszerzenia i ulepszenia są jedynie kopią tego co firma AMD ma w swoich procesorach już od dawna. Jedynie nieliczne z nich zostały dodane ponad półtora roku temu w nowej natywnej (a nie sklejaney jak u Intelu) architekturze czterordzeniowej *K10*, a w procesorach Intela nie ma ich po dziś dzień. Inną kwestią jest, że Intel celowo wycina owe rozszerzenia z wielu swoich tańszych procesorów,

również z procesorów mobilnych, a nawet z niektórych procesorów 4 rdzeniowych⁵. AMD dla odmiany nie "kaleczy" swoich procesorów, nawet najtańszych.

Biorąc pod uwagę procesory aktualnie dostępne na rynku zdecydowanie lepszym wyborem na wydajną i oszczędną platformę są procesory firmy AMD⁶, Intel może jedynie "konkurować" na konfiguracjach jedno procesorowych, ponieważ w bardziej skomplikowanych środowiskach rozwiązania Intela po prostu się nie skalują. Największym problemem procesorów Intela jest ogólnie mówiąc pamięć. *Primo* brak rozszerzeń wspierających wirtualizację pamięci w procesorze (jak chociażby *nested page table* czy też brak zintegrowanego kontrolera pamięci czego skutkiem jest o wiele mniejsza przepustowość pamięci w porównaniu do procesorów AMD. *Secundo* komunikacja pomiędzy procesorami/rdzeniami i resztą systemu przez jedynie przyspieszoną szynę FSB.

Należy też jeszcze pamiętać o następującej rzeczy, aktualnie w procesorach architektury i386 znajdują się głównie sprzętowe rozszerzenia pierwszej generacji, oznacza to niestety, że w niektórych przypadkach podejście programowe może jednak okazać się wydajniejsze (!) aniżeli rozwiązanie sprzętowe. Dotyczy to głównie rozszerzeń, które dodają tryb *VMX root* dla hypervisora, wydajność nie jest powalająca ponieważ *context switch* procesora (czyli zmiana kontekstu pracy) trwa zbyt długo. Następne generacje owych rozwiązań oraz także nowe lepsze rozwiązania bo nic w końcu nie stoi w miejscu, będą oferowały o wiele bardziej zadowalającą wydajność, mimo tych niedogodności warto z nich korzystać, gdyż różnice w wydajności nie są na tyle duże aby w jakikolwiek sposób dyskwalifikowały owe rozszerzenia.

Pamiętajmy także, że nawet jeżeli nie posiadamy sprzętu bez owych rozszerzeń nic nie stoi na przeszkodzie abyśmy używali wirtualizacji, czy to na poziomie systemu operacyjnego, czy pełnej wirtualizacji z techniką *binary translation*. Również parawirtualizacja z hypervisorem typu 1 będzie świetnie działać na takim sprzęcie.

1.8 Innowacje i nowe technologie

Wirtualizacja to aktualnie jedna z najszybciej rozwijających się dziedzin informatyki, nowe rozwiązania, lepsze pomysły, szybsze rozszerzenia sprzętowe, wielordzeniowe procesory ...

⁵Można to sprawdzić również na <http://processorfinder.intel.com>

⁶Może właśnie stąd pomysł na hasło AMD pod nazwą *Smarter Choice*.

1.8.1 Sterowniki parawirtualne

W przypadku parawirtualizacji i dostępu do kodu źródłowego systemu guest który chcemy wirtualizować wszystkie odwołania do sprzętu oraz czułe instrukcje zastąpione są w kodzie na ich odpowiedniki hypervisora, czyli odwołania *hypercall*. Sprawia to, że system guest "rozmawia" z hypervisorem bez zbędnych emulacji na czym bardzo zyskuje wydajność, która w tym wypadku jest bliska natywnej. Jeżeli używamy pełnej wirtualizacji systemu operacyjnego, którego źródeł nie posiadamy, musimy korzystać z emulowania wirtualnych urządzeń, a hypervisor musi w locie przechwytywać i zamieniać odpowiednie instrukcje. Niepotrzebnie zabiera to czas procesora oraz znacznie obniża wydajność takiego rozwiązania. Z pomocą przychodzą tutaj *sterowniki parawirtualne*, które używając wirtualnego urządzenia, komunikują się z hypervisorem za pomocą jego odwołań *hypercall*, nie ma więc potrzeby emulacji a wydajność jest w najgorszym wypadku kilkanaście procent gorsza od natywnej.

Oczywiście rozwinięciem tej idei jest wirtualizacja na sprzęcie wyposażonym w rozszerzenia do wirtualizacji operacji I/O. System guest używa wtedy zwykłych sterowników danego urządzenia, tak jakby działał samodzielnie na danej maszynie bez wirtualizacji, używając danego urządzenia natywnie z natywną wydajnością.

1.8.2 SMP w systemach guest

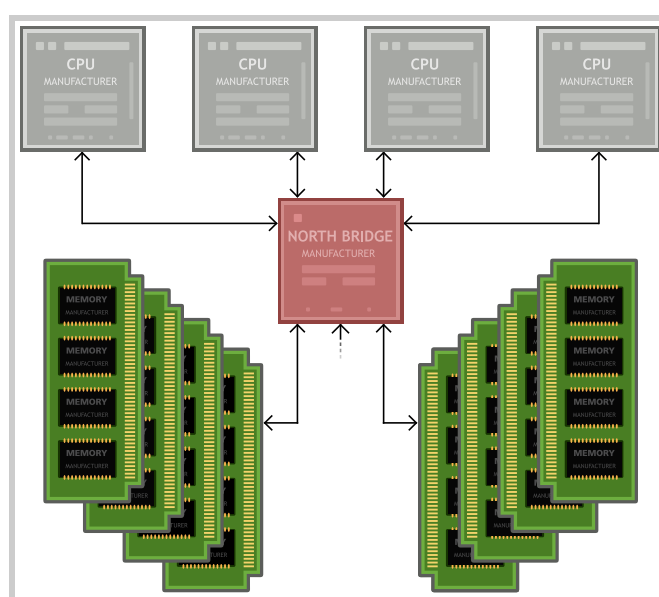
Z początku wirtualizowany system guest miał do dyspozycji jedynie jeden wirtualny procesor na którym mógł wykonywać swoje instrukcje, nawet jeżeli system host posiadał ich o wiele więcej. Zdarzało się nawet, że było to wymogiem nawet dla systemu host w niektórych przypadkach, na przykład VMware 3 na systemie FreeBSD jako host. W dzisiejszych czasach ograniczenia takie są niedopuszczalne i wywołują co najwyżej szyderczy uśmieszek.

Aktualnie używając czy to procesorów wirtualnych, czy też fizycznych, systemy guest z udostępnionymi rdzeniami są codziennością. Zarówno w rozwiązaniach serwerowych jak i przeznaczonych na stacje robocze, a na desktopie kończąc. Stawia to oczywiście nowe zadania i obowiązki przed hypervisorem, zwłaszcza jeżeli dzielimy kilka lub kilkanaście procesorów pomiędzy kilka maszyn wirtualnych.

SMP dla systemów guest jest wbrew pozorom bardzo ważnym, gdyż wirtualizowane przez nas środowiska też będą posiadały wiele wymagających aplikacji czy też serwerów, a jeden rdzeń to dla nich zdecydowanie za mało (na przykład serwer aplikacji *JAVA*), inną zaletą dowolnego przydzielania procesorów systemom guest są licencje na oprogramowanie, przykładowo *Oracle* sprzedaje licencje na swoje bazy danych w zależności od ilości procesorów w systemie, wtedy posiadając na przykład serwer z 16 procesorami, będziemy mogli używać naszej bazy danych zgodnie z licencją w wirtualizowanym środowisku 4 procesorowym.

1.8.3 NUMA

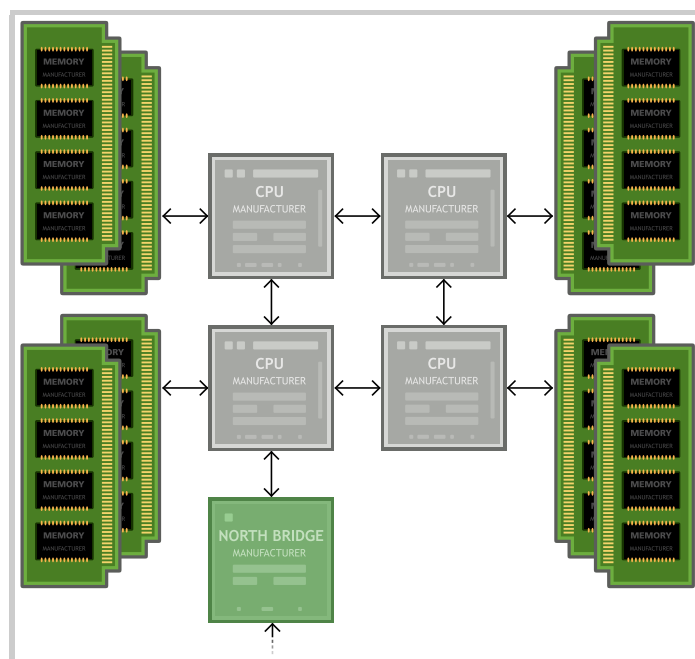
Wraz ze wzrostem liczby procesorów i rdzeni klasyczne podejście użycia szyny *FSB*, czyli podpięcie wszystkich procesorów i kości pamięci RAM, coraz bardziej pokazuje swoje wady. Niewystarczająca przepustowość, coraz większe opóźnienia wraz ze wzrostem obciążenia, czynniki znacznie wpływające na wydajność naszej platformy wirtualizacyjnej. Schemat wykorzystania szyny *FSB* przedstawia **rysunek 1.30**. Zarówno Intel jak i AMD wprowadziły już (baż zamierzają wprowadzić "na dniach") rozwiązania o wiele lepiej przystosowane do takiego obciążenia. Są to mianowicie *HyperTransport* oraz *QuickPath Interconnect*. Mimo swoich mądrych nazw nie są niczym innym jak rozwiązaniem *NUMA* czyli *Non-Uniform Memory Access*.



Rysunek 1.30: Wykorzystanie szyny *FSB* w konfiguracji SMP.

Dawniej wszystkie procesory dzieliły całą dostępną pamięć pomiędzy siebie, oczywiście używając szyny *FSB*. W architekturze *NUMA* każdy procesor posiada dedykowaną pamięć podłączoną bezpośrednio do niego, oczywiście nic nie stoi na przeszkodzie aby korzystać z pamięci innych procesorów, odbywać się to jednak wolniej niż w przypadku jego dedykowanej pamięci. Wymaganiem takiej architektury jest zintegrowany kontroler pamięci bezpośrednio w procesorze, właśnie dlatego Intel dopiero teraz wprowadza rozwiązania bazujące na *NUMA*, gdyż wcześniej nie posiadał takowego procesora w swoim portfolio, w przeciwieństwie do *AMD*, które od dawna korzysta z dobrodziejstw *NUMA*. Technologie *NUMA* przedstawia **rysunek 1.31**.

Wadą tego rozwiązania jest to, że system operacyjny musi wiedzieć jak wykorzystać architekturę *NUMA*, w przeciwnym wypadku będzie traktował dedykowane pamięci poszczególnych procesorów jako jedną spójną całość i przydzielał pamięć nie od tego procesora co trzeba, a często nawet od kilku, mimo iż pamięć dedykowana danego procesora może w skrajnym



Rysunek 1.31: Technologia *NUMA* w konfiguracji *SMP*.

przypadku nawet pozostawać pusta.

1.8.4 Cache Coherent NUMA

Pomimo swoich niezaprzeczalnych zalet architektura *NUMA* posiada również pewną niedogodność. Ponieważ mamy do czynienia z kilkoma blokami pamięci, każdy procesor przechowuje w pamięci podręcznej *cache* która część pamięci (dedykowanej czy nie) jest aktualnie używana. Tutaj pojawia się problem synchronizacji zawartości tych pamięci, tak aby wszystkie procesory "wiedziały" która część pamięci jest aktualnie dostępna, a która nie. Rozwiązanie takie nosi nazwę *Cache Coherent NUMA* (w skrócie *ccNUMA*). Dawniej zadanie optymalizacji aplikacji pod kątem *ccNUMA* oraz synchronizacji pamięci podręcznych pozostawiano programistom za pomocą *IPC*, czyli *Inter Process Communication*, co zdecydowanie nie było dobrym rozwiązaniem, nie pozwalało w pełni wykorzystać możliwości architektury *NUMA*.

Rozwiązaniem tego problemu jest sprzętowe synchronizowanie tej części *cache* procesorów, która odpowiada za obraz pamięci, oczywiście za pomocą specjalnego układu. Rozwiązania takie stosuje się na przykład w procesorach *opteron* firmy *AMD*. Aby więc hypervisor mógł efektywnie wykorzystać dany sprzęt musi być świadom czy działa na architekturze *NUMA*.

1.8.5 Topology Aware Scheduler

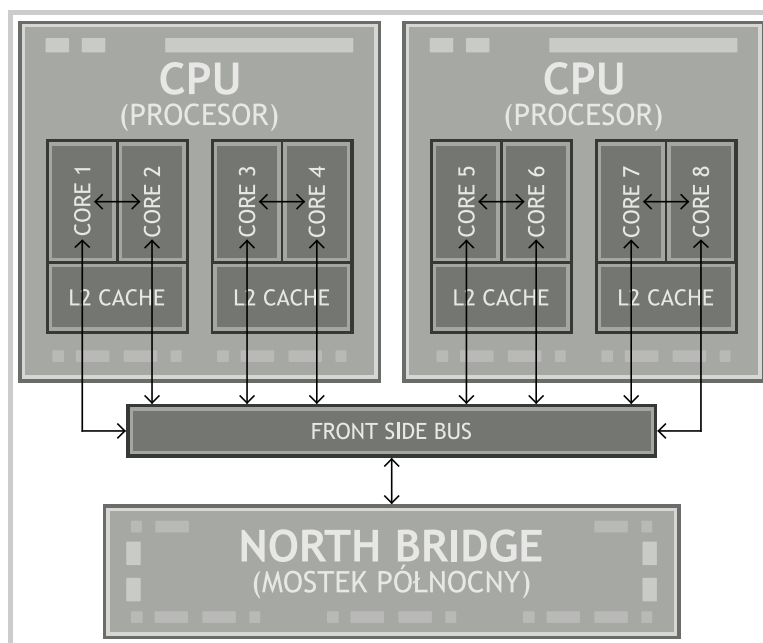
Skoro "nauczyliśmy" już hypervisor efektywnego działania na architekturze NUMA, czas iść o krok dalej i zająć się topologią sprzętu na którym hypervisor będzie pracował. Jak ktoś kiedyś mądrze stwierdził porównując procesory AMD i Intela, z pułapu 10 000 stóp wyglądają bardzo podobnie, jednak jak to zwykle bywa diabeł tkwi w szczegółach. Mimo iż obydwie korporacje mają w swojej ofercie procesory 4 rdzeniowe znacząco różnią się one budową.

Intel jako pierwszy wprowadził na rynek taki procesor pod nazwą *Core 2 Quad* (*Xeon*⁷ na rynku serwerowym) i tak naprawdę są to tylko 2 sklejone procesory architektury *Core 2 Duo*, nic więcej. Rozwiązanie takie ma oczywiście swoje plusy i minusy. Z początku na pewno będzie tańsze w produkcji, gdyż nie musimy projektować nowego procesora prawie "od zera", a cała infrastruktura już produkuje potrzebne komponenty. Z drugiej strony wydajność komunikacji między rdzeniami zależy od tego które dwa rdzenie ze sobą "rozmawiają". Ma to bardzo duże znaczenie przy rozdzielaniu wątków przez *scheduler* czyli *algorytm szeregowania*. Przeważnie nie zna on budowy procesorów i wszystkie rdzenie traktuje z jednakowym priorytetem, co znacznie ułatwia jego implementację, cierpi jednak na tym wydajność. Nie jest to tak istotne gdy musimy obsłużyć jeden taki procesor, wtedy prosta implementacja *schedulera* zdaje egzamin. Schody zaczynają się dopiero w konfiguracjach wieloprocessorowych.

Rozwiązaniem jest tutaj *topology aware scheduler*, czyli algorytm szeregujący, który ma zaimplementowane różnice w budowie procesorów. Wie, że w przypadku *Core 2 Quad* każde 2 rdzenie posiadają wspólną pamięć *cache* L2, wie też, że kolejne 2 rdzenie (nieparzysty oraz parzysty) będą rozmawiać, ze sobą bezpośrednio, a każde 2 rdzenie (parzysty oraz nieparzysty) będą rozmawiały przez szynę *FSB*. Będzie też musiał wiedzieć, że rdzenie które nie rozmawiają ze sobą bezpośrednio, mają takie same opóźnienia do innych rdzeni nie rozmawiających bezpośrednio, niezależnie od procesora. Kolejnym aspektem jest wspólna pamięć *cache*, o wiele bardziej opłaca się rozdzielać wątki na 2 rdzenie, które rozmawiają ze sobą bezpośrednio i posiadają wspólną pamięć podręczną, pomijając korzyści z większej wydajności dochodzi tutaj jeszcze kwestia lepszego wykorzystania pamięci *cache*, gdyż gdyby wątki zostały przydzielone na rdzeniach nie rozmawiających ze sobą bezpośrednio, to informacje w pamięci podręcznej byłyby zdublowane w dwóch miejscach. Jest to bardzo nieefektywne, zwłaszcza że pamięć ta jest małych rozmiarów i do tego dosyć droga w produkcji. Przykładem implementacji takiego *schedulera* jest *ULE* z systemu *FreeBSD*. Schemat sposobu komunikacji rdzeni w procesorach *Core 2 Quad* przedstawia **rysunek 1.32**.

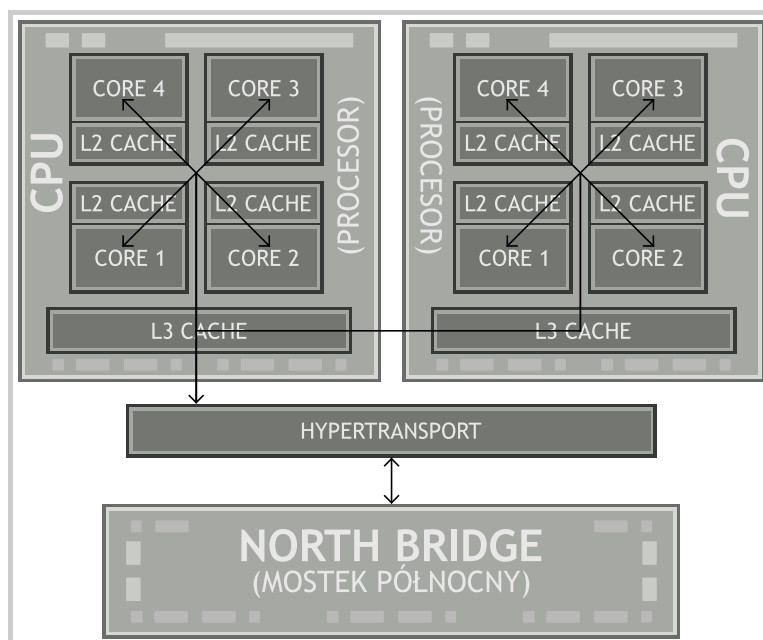
Zupełnie inaczej jest w przypadku procesorów AMD. Natywna konstrukcja pozwala na szybka komunikacje wszystkich 4 rdzeni w obrębie procesora, nie ma też potrzeby przydzielania wątków parami, gdyż każdy rdzeń posiada swoją własną pamięć *cache* L2, a do dyspozy-

⁷Jedyna różnica pomiędzy procesorami *Core 2* oraz *Xeon* polega na tym, że te pierwsze mają programowo zablokowaną możliwość pracy w konfiguracjach SMP w *firmware* procesora.



Rysunek 1.32: Sposób komunikacji rdzeni w procesorach *Core 2 Quad*.

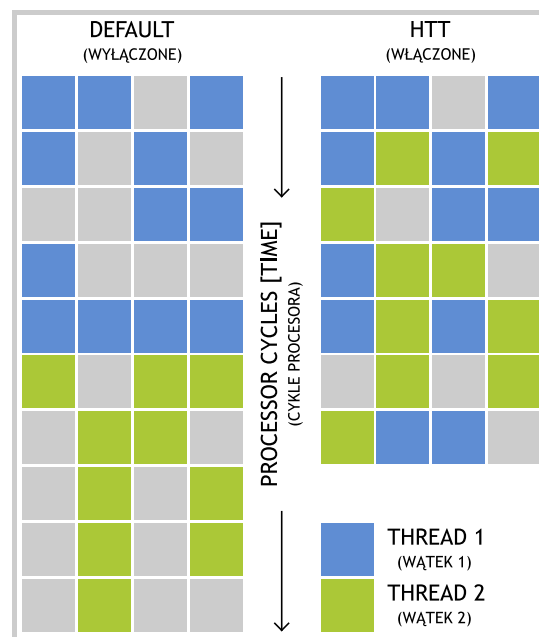
cji wszystkich 4 rdzeni jest jeszcze dodatkowa współdzielona pamięć L3. Jedyne opóźnienia więc jakie występują, to opóźnienia w komunikacji pomiędzy rdzeniami różnych procesorów. Schemat komunikacji rdzeni w procesorze AMD o architekturze *K10* przedstawia **rysunek 1.33**.



Rysunek 1.33: Sposób komunikacji rdzeni w procesorach architektury *K10*.

Scheduler hypervisora powinien też wiedzieć o logicznych procesorach, stosowanych w

procesorach Intel'a z technologią *HTT* czyli *Hyper Threading Technology*. Pozwala ona używać jednego fizycznego rdzenia jako dwóch logicznych w celu dociążenia układów procesora, gdyż często jest tak, iż wiele jednostek *CPU* nie jest wykorzystywanych w tym samym czasie. Gdy jeden wątek korzysta z jednostki *FPU* inny może korzystać z układu *ALU* i odwrotnie, podobnie z rejestrami procesora. Algorytm szeregowania procesów, który nie jest świadom takiego rozwiązania, będzie obciążał logiczne rdzenie tak samo mocno jak fizyczne, co prowadzi do spadków wydajności. Rozwiązanie *HTT* w praktyce przedstawia **rysunek 1.34**.



Rysunek 1.34: Jednostki wykonawcze procesora z włączoną/wyłączoną obsługą *HTT*.

1.8.6 Zarządzanie energią i skalowanie

Poza wydajnością bardzo ważne znaczenie ma również pobór mocy danej platformy, bo co z tego, że osiąga ona bardzo dobrą wydajność, jeżeli na jej utrzymanie wydajemy krocie. Nie tylko wirtualizacji coraz bardziej zaczyna się liczyć współczynnik *performance per watt*, czyli wydajność jaką osiągniemy przy danym zużyciu energii, im taki wskaźnik wyższy tym lepiej oczywiście.

Jedną z metod jest skalowanie prędkością i napięciem procesorów czy też ich rdzeni. Rozwiązanie takie skutecznie obniża zarówno zużycie prądu jak i temperaturę wewnątrz komputera. Pytanie tylko "komu" tak naprawdę pozostawić zarządzanie, czy hypervisor ma decydować na ile system guest obciąża procesory i je skalować, czy też dostarczyć systemowi guest standardowe mechanizmy skalowania procesora i to właśnie jemu pozostawić zarządzanie. Zależy to od tego jak przydzielamy procesory maszynom wirtualnym. Jeżeli dany system guest będzie posiadał na wyłączność jeden lub kilka rdzeni tego samego procesora, czy też kilka proce-

sorów, wtedy teoretycznie sterowanie owym mechanizmem możemy pozostawić systemowi guest. Jeżeli natomiast dzielimy procesor czy też rdzeń na kilka maszyn wirtualnych, wtedy lepszym rozwiązaniem będzie pozostawienie decyzji i sterowania hypervisorowi.

Mimo wszystko lepszym podejściem będzie chyba jednak pozostawienie nad tym mechanizmem pełnej kontroli hypervisorowi, jest tutaj kilka mocnych argumentów za. Nie wszystkie systemy operacyjne obsługują skalowanie procesora, jeżeli będzie za to odpowiadać hypervisor, wówczas mamy pewność, że tak czy tak procesory będą się odpowiednio skalować, nawet w przypadku gdy jeden procesor, czy też rdzeń jest przydzielony więcej niż jednemu systemowi guest. Do tego dochodzi również możliwość "gaszenia" zarówno rdzeni jak i całych procesorów, na przykład na systemach, które "nudzą się" przez kilka godzin na dobę, będzie to prowadzić do kolejnych oszczędności. Na systemach o małym obciążeniu hypervisor mógłby na przykład przenieść wszystkie procesy na 1 rdzeń, a pozostałe wyłączyć, do tego zwolnić i obniżyć napięcie dla tego ostatniego, jeżeli obciążenie będzie wystarczająco małe.

Pozostaje jeszcze kwestia sprzętowa, procesory muszą obsługiwać te rozwiązania, zarówno skalowanie jak i usypianie czy też wyłączanie. Obecnie procesory zarówno serwerowe jak i te przeznaczone na desktop obsługują zarówno skalowanie, obniżanie napięcia jak i wyłączanie poszczególnych rdzeni, pozostaje więc odpowiednia implementacja tych rozwiązań, tak aby hypervisor możliwie najefektywniej wykorzystywał sprzęt.

1.8.7 Sprzętowa akceleracja grafiki

Wiele aplikacji do poprawnego działania wymaga sprzętowej akceleracji grafiki 3D, do niedawna maszyny wirtualne oferowały jedynie programowe wyświetlanie obrazu, niestety tylko w 2D ale nawet wtedy zbyt wolno by pozwolić na komfortową pracę. Używanie czegokolwiek w trzech wymiarach po prostu mijało się z celem.

Wirtualizacja układu *GPU* jest co najmniej problematyczna, z kilku powodów. W większości przypadków mamy dostęp jedynie do binarnych sterowników kart graficznych, nie mamy więc najważniejszego, czyli kodu źródłowego sterownika. Kolejną przeszkodą jest to, iż karty graficzne z różnych generacji posiadają różne niekompatybilne interfejsy [[de Lara / H. A. Lagar-Cavilla, 2006](#)]. Nie posiadamy też specyfikacji samych układów. Co prawda ostatnio wiele się zmieniło w tym temacie *in plus* dzięki temu, że firma AMD wydała dokumentację 2D swoich nowszych układów z serii *R600* bez potrzeby podpisywania *NDA*⁸ przez deweloperów, dokumentacja 3D ma zostać opublikowana w najbliższym czasie. Na płaszczyźnie otwartości najlepszym wyborem nadal pozostaje Intel ze swoimi otwartymi sterownikami. Dostarczył także pełną dokumentację układów *965G* oraz *G35*, opisuje ona kompletną specyfikację 2D oraz 3D, a nawet dokładny opis akceleracji wideo. Powoli swoje sterowniki otwiera też VIA,

⁸ *Non Disclosure Agreement* to umowa poufności często wymagana przy otwieraniu specyfikacji sprzętu.

jednak nie można tutaj na razie mówić o tak dużych porcjach danych jak w przypadku AMD. Na szarym końcu nadal pozostaje nVidia, która nie dostarcza żadnego wsparcia programistom, ani w postaci dokumentacji, ani w postaci otwartego sterownika⁹.

Małym światełkiem w tunelu jest tutaj projekt *Nouveau*, który podjął się jakże trudnego zadania napisania otwartych sterowników dla kart nVidii od zera, na bazie metody podobnej do *reverse engineering*, podobnej gdyż EULA kart nVidia dodatkowo zabrania takiego poznawania działania sterowników.

VMGL

Jednym z rozwiązań tego problemu jest przechwytywanie i wykonywanie instrukcji *OpenGL* z systemu guest bezpośrednio na karcie graficznej systemu host, a następnie zwrócenie wyników do systemu guest. Wielkim plusem takiego rozwiązania jest jego prostota, oraz efektywność, gdyż nie ma tutaj miejsca żadna emulacja, podwójne kopiowanie, czy też binarna translacja w locie. Jedyne opóźnienie polega jedynie przekazaniu odpowiednich instrukcji *OpenGL* z systemu guest do karty graficznej systemu host i z powrotem. Rozwiązuje to problem zarówno binarnych sterowników jak i jednego spójnego interfejsu, gdyż wszystkie karty z ostatnich lat w pełni obsługują standard *OpenGL*. Niestety wadą tegoż rozwiązania jest to, iż działa tylko dla aplikacji 3D, które korzystają z ... *OpenGL*. W przypadku aplikacji używających zamkniętych interfejsów (czyli *proprietary*) pokroju *Direct3D*, nic nie możemy zrobić.

Takie właśnie podejście oferuje technologia *VMGL*, pozwala na wieloplatformową wirtualizację obrazu 3D za pomocą instrukcji *OpenGL*. Jest niezależna zarówno od hypervisora, systemów guest jak i konkretnych modeli kart graficznych. *VMGL* do przekazywania instrukcji systemu guest używa *WireGL*, które posiada kilka ważnych rozwiązań znacznie poprawiających wydajność. Instrukcje *OpenGL* są agregowane i przesyłane "paczkami" a nie pojedynczo, a obliczenia dotyczą tylko miejsc na ekranie, które uległy zmianie. Dodatkowo przekształcenia nie są nakładane jedno po drugim, ale łączone są w jedno przekształcenie i dopiero nakładane. Rozwiązanie to dostępne jest dla praktycznie każdej liczącej się na rynku technologii wirtualizacji: *Xen* (HVM / parawirtualizacja), *Virtualbox*, *QEMU*, *KVM* czy nawet *VMware*.

GPGPU

Renderowanie grafiki to nic innego jak wysyłanie danych do obliczeń i zwrócenie wyniku na ekran, różnica polega tylko na tym, którym interfejsem owy wynik otrzymamy. Może to być *OpenGL*, *Direct 3D* czy nawet technologia *GPGPU* (*General Purpose Computing on Graphics Processing Units*). Nic nie stoi na przeszkodzie, aby właśnie ten ostatni mechanizm wykorzystać w przypadku wirtualizacji systemu guest i za jego pomocą generować obraz 3D. Rozwiązanie

⁹Dostępny jest jedynie sterownik dla serwera X11 oferujący podstawową obsługę 2D.

takie może być realizowane przez wirtualne urządzenie, które hypervisor wystawi systemowi guest, system guest zaś będzie wymagał odpowiedniego sterownika. Hypervisor zaś będzie odbierał żądania wirtualnej karty graficznej, wysyłał je do prawdziwej karty graficznej, a rezultat wysyłał do systemu guest.

Rozwiązanie takie jest bardzo elastyczne, gdyż karty graficzne posiadają pewną ilość zunifikowanych jednostek obliczeniowych *unified shaders*, te z najwyższej półki posiadają ich setki. Przydzielając systemowi guest taką wirtualną kartę graficzną, możemy zdecydować ile takich jednostek użyć, co więcej, możliwe byłoby też bardzo wygodne zarządzanie takimi jednostkami, tak samo jak w przypadku rdzeni procesorów przydzielanych maszynie wirtualnej. Wymagającemu środowiskowi będzie można przydzielić sporo jednostek na sztywno, podczas, gdy innym mniej wymagającym graficznie systemów, przydzielić pewną pulę jednostek do podziału. Rozwiązanie takie nie jest aktualnie zaimplementowane, ale nic nie stoi na przeszkodzie aby takowe wdrożyć.

Jest wiele dostępnych rozwiązań do realizowania obliczeń *GPGPU*, *CUDA* jest rozwiązaniem stosowanym przez koncern *nVidia* w swoich kartach graficznych. Firma *nVidia* starała się także zachęcić swego największego konkurenta na polu kart graficznych czyli *ATI* do implementacji tejże technologii. Ten ostatni na razie jednak nie zdecydował się na ten krok, prawdopodobnie ze względów finansowych, gdyż *nVidia* na pewno będzie wymagać odpowiednich opłat licencyjnych za korzystanie z owego rozwiązania.

Firma *ATI* również chciała używać własnego rozwiązania pod nazwą *Stream* jednak zrezygnowano z niego na rzecz otwartego języka *OpenCL*, czyli *Open Computing Language*. Mechanizm ten wykorzystuje rozwiązania takie jak *OpenGL* (do grafiki) oraz *OpenAL* (do dźwięku). Obecnie *OpenCL* został zgłoszony do procesu standaryzacji, co po pozytywnym zatwierdzeniu przez organizację *ISO* z pewnością zwiększy jego konkurencyjność pośród innych rozwiązań.

Wadą tego rozwiązania jest z pewnością przywiązanie się do układów tylko jednego producenta, niezależnie czy wybierzemy *CUDA* czy *OpenCL*. Ważne jest, aby w końcu firmy doszły do porozumienia i wspierały jeden wspólny standard, najlepiej *OpenCL* gdyż jest otwarty, i korzysta z otwartych rozwiązań. Firma *nVidia* na pewno nie zrezygnuje ze swego rozwiązania *CUDA*, więc jest też szansa, że i *ATI* z czasem jednak skusi się na jej implementację, chociaż nie jest to nic pewnego.

Warto także wspomnieć o niedawnym prototypie karty graficznej Intel'a o nazwie kodowej *Larrabee*. To w pełni programowalny¹⁰ układ wyposażony w rdzenie architektury *i386* (docelowo od 8 do 48 rdzeni), swego rodzaju hybryda *CPU* i *GPU*. Obsługuje standardowe API jak *OpenGL* czy *DirectX*, ale pozwala także na *ray tracing* czy obliczenia fizyczne (inaczej *physics processing*). Oczywiście głównym przeznaczeniem tego układu są obliczenia *GPGPU*

¹⁰ Aktualne generacje kart graficznych są tylko w pewnym stopniu programowalne.

i bezpośrednia konkurencja dla kart graficznych nVidii i ATI.

Gallium3D

Rozwiązanie *Gallium3D* to nowa biblioteka graficzna tworzona przez firmę *Tungsten Graphics*. Dostarcza ona wielu ciekawych mechanizmów, upraszcza pisanie sterowników oraz implementacje nowych API. Posiada też bardzo ciekawe rozwiązanie pozwalające "podglądać" odwołania pomiędzy sterownikiem a sprzętem. Pozornie nie brzmi to zbyt powalająco, jest to jednak bardzo użyteczna funkcjonalność, użyteczna nie tylko przy projektowaniu i pisaniu sterowników. Można ją wykorzystać w wirtualizacji do przechwytywania zapytań systemu guest do karty graficznej, a potem wykonywać je bezpośrednio na karcie graficznej systemu host, zwrócenie wyników będzie już wtedy jedynie formalnością. Biblioteka *Gallium3D* będzie wykorzystywana do wirtualizacji GPU w następnej wersji hypervisora *Xen 3.4*.

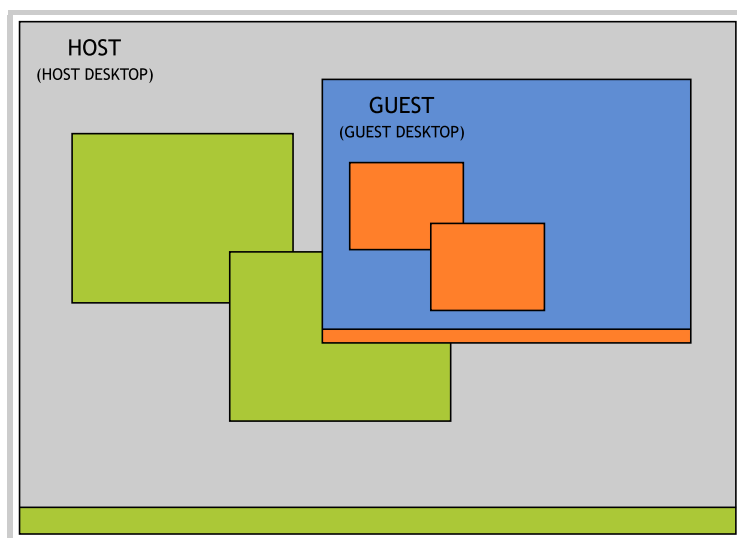
Sprzętowa wirtualizacja

Na koniec istniejące i działające rozwiązanie, polegające na wykorzystaniu sprzętowych rozszerzeń do wirtualizacji operacji I/O, czyli *VT-d* lub *IOMMU*. Możemy zarówno "oddąć" całą kartę graficzną pod kontrolę systemu guest, jak i po prostu dzielić ją pomiędzy wieloma systemami. Hypervisor musi jednak mieć zaimplementowaną obsługę owego rozszerzenia sprzętowego.

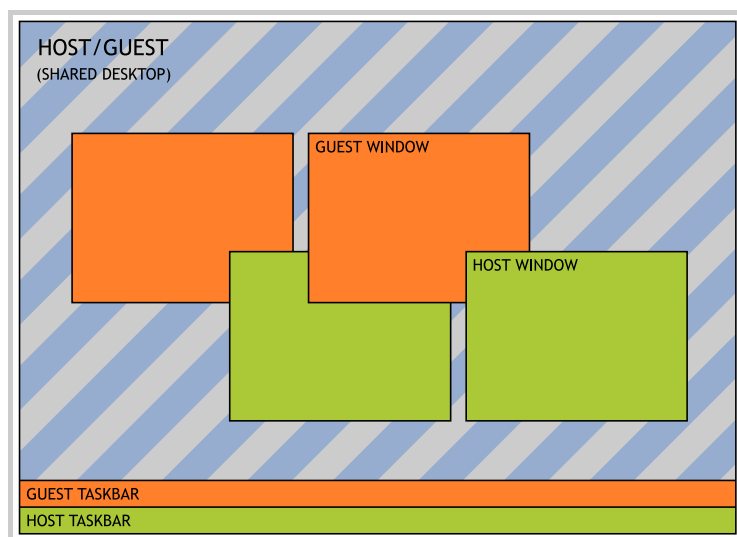
1.8.8 Seamless Mode

Rozwiązanie stosowane na stacjach roboczych i desktopach. Dotychczas system guest uruchamiany był w swoim okienku, w czymś na miano wirtualnego monitora, okno to jak każde inne podlegało menadżerowi okien danego środowiska, w którym uruchamialiśmy system guest. Bardzo utrudniona była wymiana plików pomiędzy systemem guest i host, korzystano na przykład z samby i montowania po sieci katalogu w systemie guest. I mechanizm *drag and drop* można było co najwyżej pomarzyć, podobnie jak o wygodnej pracy w takim środowisku. To tradycyjne podejście przedstawia **rysunek 1.35**.

Tryb *seamless mode* (zamiennie zwany również *coherence*) to tryb wirtualizacji, w którym aplikacje uruchomione w systemie guest, zachowują się tak jakby były aplikacjami uruchomionymi w systemie host. Podlegają menadżerowi okien systemu host, działa mechanizm *drag and drop*, a dzielenie plików jak i zwykła praca jest o wiele wygodniejsza. Rozwiązanie to przedstawia **rysunek 1.36**.



Rysunek 1.35: Wirtualizacja systemu guest w oknie.



Rysunek 1.36: Wirtualizacja w trybie *seamless mode*.

Rozwiązanie to przeważnie działa tylko w "jedną stronę", mianowicie implementuje się ten tryb dla wirtualizacji systemu *Windows*, tak aby można byłoby go wygodnie wirtualizować na innych systemach operacyjnych. Spowodowane jest to tym, iż sporo specjalistycznego oprogramowania dostępnego jest na "jeden słuszny system operacyjny" a producenci często nie widzą potrzeby portowania swoich aplikacji na inne systemy, lub też dostarczone przez nich programy nie posiadają pełnej funkcjonalności oryginalnego programu. Swego rodzaju wyjątkiem jest tutaj wirtualizacja systemu *Windows XP* na systemie *Windows Vista* jest to jednak spowodowane sporą wadliwością i nieprzewidywalnością tego ostatniego, ale spora część producentów nie widzi już potrzeby wydawania sterowników na starszą odmianę systemu z Redmond, a na nowszy nie ma dostępnej naszej aplikacji, bądź też praca z nią jest w nim niemożliwa.

Możliwa jest też ograniczona realizacja tego mechanizmu przez protokół *RDP* (czyli *Remote Desktop Protocol*), po prostu łączymy się z wybranym komputerem, czy też maszyną wirtualną i uruchamiamy na niej wybraną aplikację, wynik otrzymujemy w postaci okna samej aplikacji na naszym pulpicie. Możemy jednak zapomnieć o *drag and drop*.

1.9 Zalety

Średnie obciążenie serwera sięga około 5-10%, oznacza to, że przez większość czasu serwer po prostu nic nie robi, lub robi bardzo niewiele. Używanie większej ilości serwerów z takim obciążeniem jest po prostu nieopłacalne, wyższe rachunki za energię elektryczną czy dodatkowe obciążenie klimatyzacji, a także większe koszty samego sprzętu. Zamiast kupować kilka fizycznych serwerów które "będą się nudzić" możemy kupić jeden mocniejszy i na nim za pomocą wirtualizacji postawić potrzebne systemy. Bardzo zwiększa to **utyliczację i konsolidację** naszego sprzętu, a **koszty** funkcjonowania są o wiele mniejsze. Oznacza to także oszczędności w innych aspektach centów danych, mniej komputerów oznacza mniej potrzebnego miejsca, znacznie mniejsze zapotrzebowanie na klimatyzację. Należy jednak pamiętać, że na jednym fizycznym serwerze mogą komfortowo współdziałać usługi, które nie wymagają tych samych zasobów w tym samym czasie. Musimy więc odpowiednio rozłożyć nasze usługi pomiędzy dostępny sprzęt aby maksymalnie go wykorzystać.

Mając uruchomionych wiele usług na jednym serwerze, możemy stanąć w sytuacji, gdy jedna z usług przez swoje tymczasowe wadliwe działanie odbiera czy to zasoby, czy też w ogóle uniemożliwia działanie pozostałym usługom w naszym systemie. Za pomocą wirtualizacji możemy od siebie **odseparować** wszystkie nasze usługi, przez co jakakolwiek awaria jednej z usług nie pociągnie za sobą awarii pozostałych usług.

Wirtualizacja, pozwala na o wiele **wygodniejsze zarządzanie** całą naszą infrastrukturą, im większa infrastruktura, tym bardziej docenimy ową wygodę. Cały nasz serwer, czy też usługa może znajdować się w jednym pliku na dysku. Możemy go przenieść na inny serwer i tam uruchomić, mimo iż będzie tam zupełnie inny sprzęt (a czasami nawet zupełnie inna architektura). Bardzo ułatwia to także robienie kopii zapasowych.

Jeżeli korzystamy z odpowiednich rozwiązań to możemy **przenieść działającą maszynę wirtualną** na inny serwer, bez przerywania jej pracy. Możemy także korzystać ze *snapshotów* naszych maszyn, czyli uruchomić usługę z pewnego określonego miejsca w czasie, na przykład z poprzedniego tygodnia. Innym bardzo użytecznym rozwiązaniem jest **zatwierdzanie zmian**

w momencie, gdy jesteśmy zadowoleni ze zmian wprowadzonych do naszego serwera czy też usługi, czyli tak zwany *commit*. Jeżeli nie jesteśmy zadowoleni, lub po prostu coś poszło nie tak, uruchamiamy maszynę wirtualną naszego serwera i mamy stan sprzed kłopotliwych zmian.

Wirtualizacja jest również świetnym rozwiązaniem do **debugowania** nowych rozwiązań, jądra systemu operacyjnego czy nawet całych klastrów. Dotyczy to także testowania nowych usług czy serwerów, możemy bardzo łatwo i szybko stworzyć odpowiednie testowe wirtualne środowisko, bez wpływu na pracę pozostałych naszych usług czy serwerów. Możemy więc dowolnie testować nowe rozwiązania bez obawy, niezależnie od tego czy są bezpieczne czy nie.

Nic nie stoi na przeszkodzie aby używać wielu różnych systemów operacyjnych na jednym serwerze, Windows, FreeBSD, Solaris, czy jakiegokolwiek inny system który przyjdzie nam do głowy. Podobnie z poszczególnymi rozwiązaniami, które są dostępne na wybrane platformy, nie jesteśmy już więcej "przywiązani" do jednego rozwiązania.

Dotychczas aby przetestować jakieś rozwiązanie, potrzeba było całego serwera aby nie było zagrożenia "uszkodzenia" pozostałych usług. Trzeba było się upewnić, że dostawiając kolejny serwer będzie wystarczająco chłodzenia i prądu, potem oczywiście zainstalować tam odpowiedni system operacyjny i dopiero rozpocząć testowanie naszego nowego rozwiązania czy też usługi. Jeżeli nie posiadamy odpowiedniego sprzętu na nowy serwer należy oczywiście jeszcze doliczyć czas na dostawę. Używając wirtualizacji zadanie to sprowadza się do minut, jest tylko kwestią podania przez nas ilu zasobów przydzielimy nowej "maszynie" testowej i jaki będzie miała adres IP, po chwili możemy już testować nasze nowe rozwiązania.

Awaryje mają to do siebie, że przytrafiają się w najmniej odpowiednim momencie, dzięki wirtualizacji dana usługa może zostać przywrócona automatycznie do pracy, bez potrzeby ściągnięcia administratorów w środku nocy w celu naprawienia i przywrócenia usług, dzięki wirtualizacji mogą się tym spokojnie zająć następnego dnia.

Dzięki wirtualizacji nasze usługi będą posiadały większe stopień *high availability*, gdyż w przypadku gdy jeden z naszych serwerów padnie, czy też będziemy widzieć, że dzieje się z nim coś nie tak, możemy przenieść nasze usługi na inny fizyczny serwer. Możemy także uruchomić kilka takich samych instancji naszych usług na kilku fizycznych serwerach zapewniając sobie pełną ochronę przed takimi awariami, na przykład za pomocą protokołu *Virtual Router Redundancy Protocol*.

Jeżeli ktoś włamie nam się do jednej z naszych maszyn wirtualnych, to tylko do niej, stracimy jedną usługę a nie kilka czy kilkanaście, do tego w każdej chwili możemy przywrócić naszą usługę w praktycznie kilka minut, kwestia podmiany obrazu z maszyną wirtualną i jej uruchomienie.

W przypadku zamkniętych aplikacji, których potrzebujemy do pracy, a nie są one dostępne na naszej platformie, nie musimy już męczyć się w *dual boot* czyli restartować komputer w celu uruchomienia innego systemu operacyjnego, teraz z trybem *seamless* możemy po prostu wygodnie pracować na dwóch systemach jednocześnie.

1.10 Wady

Zależnie od wykorzystanego przez nas rozwiązania, możemy spodziewać się pewnego **spadku wydajności**, w stosunku do pojedynczej maszyny, jednak w przypadku serwerów których średnie obciążenie sięga 5-10% nie powinno mieć to większego znaczenia. W dodatku wirtualizacja na poziomie systemu operacyjnego zapewnia tą samą wydajność co natywnie, gdyż korzysta bezpośrednio z jądra systemu host. Również parawirtualizacja zapewnia prędkość bliską natywnej, tak więc spadki wydajności są zależne od stosowanego przez nas rozwiązania.

Architektura i386 nie była projektowana z myślą o wirtualizacji, a aktualne rozszerzenia sprzętowe nie zawsze zapewniają odpowiednią wydajność. Inną wadą jest, że aby skorzystać z wielu metod wirtualizacji musimy zakupić sprzęt wyposażony w odpowiednie rozszerzenia sprzętowe do wirtualizacji.

Wirtualizacja jest bardzo dobrym rozwiązaniem i w pewnych sytuacjach po prostu nie opłaca się jej nie stosować, ważne jednak jest, aby cała infrastruktura wirtualizacji została odpowiednio zaprojektowana, przez kogoś kto posiada **odpowiednie doświadczenie** w tym temacie. Może się przecież zdarzyć, że zajmie się tym zadaniem osoba niewykwalifikowana, dopiero zaczynająca pracę z wirtualizowanymi środowiskami i popełni wiele błędów projektując wirtualną infrastrukturę, co niestety zrobi więcej złego niż dobrego, należy więc dopisać do kosztów zdobycie doświadczenia i odpowiedniej wiedzy na temat wirtualizacji aby ją efektywnie i bezpiecznie stosować.

1.11 Zastosowania

Wirtualizacja ma tak wiele zastosowań, że ogranicza nas praktycznie tylko wyobraźnia i pomysłowość. Na rynku istnieje bardzo wiele wyspecjalizowanych maszyn wirtualnych, zarówno przeznaczonych na serwery, stacje robocze jak i zwykły desktop. Rozwiązania *open source* oraz *proprietary* przeznaczone do użytku domowego i korporacyjnego. Daje to cały wachlarz różnych możliwych zastosowań, w zależności od potrzeb.

Przede wszystkim konsolidacja. Mając do dyspozycji kilkanaście, a czasem nawet kilkadziesiąt rdzeni, wirtualizacja umożliwia nam najefektywniejsze wykorzystanie takiej mocy. Będziemy mogli podzielić zasoby takiego systemu pomiędzy wielu klientów, zupełnie tak jakby pracowali na kilku niezależnych maszynach, każdy może mieć zupełnie inny system operacyjny, z innym zestawem oprogramowania, często do zupełnie innych zastosowań niż pozostali klienci. Na przykład jeden z klientów będzie miał przydzielone 4 rdzenie i 2 dedykowane karty sieciowe, gdyż serwuje strony www z bazą danych, inny klient będzie korzystał z karty sieciowej i procesorów dzielonych pomiędzy pozostałymi klientami, a jeszcze kolejny klient będzie miał dedykowane 8 rdzeni, gdyż jest developerem Java i potrzebuje dużej mocy obliczeniowej.

Innym zastosowaniem wirtualizacji jest testowanie i pisanie nowych rozwiązań, zwłaszcza tych niskopoziomowych związanych z systemami operacyjnymi, klastrami i sieciami. Nic nie stoi na przeszkodzie abyśmy uruchomili 16 identycznych maszyn wirtualnych w celu testowania wydajności oprogramowania klastra uruchomionego na tychże 16 systemach. Wirtualizacja zapewnia tutaj duże oszczędności na wielu szczeblach. Bez wirtualizacji musielibyśmy posiadać 16 fizycznych maszyn, których koszt byłby z pewnością niemały, co prawda maszyna na której będziemy testować nasz klaster również powinna posiadać odpowiednio większą moc, ilość rdzeni i pamięci, jednakże ceny tych podzespołów zeszły już do takiego poziomu, że i tak będzie to bardziej opłacalne niż wiele słabszych komputerów.

Drastycznie obniża to pobór mocy naszego zestawu testowego, inna sprawa gdzie trzymać taką ilość komputerów, a jedyną sensowną odpowiedzią wydaje się być profesjonalna serwerownia. Dochodzi też kwestia wygody testowania tych rozwiązań, jeżeli jeden z serwerów składających się na ten klaster się zawiesi, musimy się do niego przejść i go zrestartować, a dzięki wirtualizacji wystarczyłoby zapewne jedna komenda. Nie wspominając już o tym, że system uruchomiony w maszynie wirtualnej uruchomi się o wiele szybciej niż na fizycznym komputerze, gdzie wiele czasu tracimy na sam *BIOS POST*. Możemy też się ograniczyć do wyjmowania odpowiednich skrętek ze *switcha*, ale dalej będzie nas to odciągało od samego procesu testowania, a wirtualizacja i tak załatwi to lepiej i efektywniej.

Inną zaletą wykorzystania wirtualizacji jest z pewnością fakt, że możemy przetestować każde z dostępnych rozwiązań bez obawy o bezpieczeństwo naszej infrastruktury, gdyż je-dyne szkody (jeżeli w ogóle się przydadzą) zostaną wyrządzone wewnątrz maszyny wirtual-

nej, czyli nie mają tak naprawdę żadnego znaczenia.

Załóżmy, że bliżej nieokreślony czas temu zmieniliśmy system na Linux, znaleźliśmy już zamienniki dla naszych ulubionych programów, z którymi byliśmy przyzwyczajeni pracować na co dzień na platformie Windows. Jest nam tu dobrze, mamy co chcemy i możemy pracować. Brakuje nam jednak jednego programu, akurat tego ulubionego, lub po prostu dla nas kluczowego. Wiele osób decyduje się na *dual boot* z systemem Windows pozostawionym na jednej z pozostałych partycji dysku w celu posiadania dostępu tej jednej aplikacji.

Zawsze to jakieś rozwiązanie, jednak jest ono problematyczne i nieefektywne, pomijając już nasze delikatnie mówiąc niezadowolenie z oczekiwania na przeładowanie systemu, czy to w jedną czy w drugą stronę. Innym wyjściem może być używanie implementacji API systemu Windows na systemach UNIX, którą oferuje projekt WINE, jednak mimo iż projekt ten dynamicznie rozwija się od ponad 15 lat to niestety nie zapewnia on jeszcze pełnej zgodności. Tutaj właśnie przydaje się wirtualizacja, przez uruchomienie całego systemu Windows zapewni nam natywne środowisko pracy dla naszej aplikacji, a dzięki współdzieleniu plików z naszym systemem host zapewni nam łatwą wymianę plików na których pracujemy. Do tego zaoszczędzimy sporo czasu, gdyż nie będzie już konieczności przeładowywania systemów.

Jedną z zalet hypervisoru typu 1 (jak na przykład *Xen*) jest ukrywanie skomplikowanej architektury sprzętu przed systemami guest. Istnieją systemy operacyjne, które mają problemy z efektywnym rozdzielaniem zadań na więcej niż 4 procesory lub rdzenie. Monitor *Xen* może udostępnić systemom guest po jednym wirtualnym procesorze a sam będzie sobie rozkładał obciążenie na fizyczne procesory. Możemy tutaj oczywiście wymieniać jeszcze długo, gdyż zastosowań wirtualizacji jest mnóstwo, ale niestety nie mamy miejsca na zaprezentowanie każdego z nich.

Rozdział 2

Dostępne maszyny wirtualne

2.1 Hypervisor typu 1

2.1.1 Xen



<http://xen.org>

Monitor *Xen* to hypervisor typu *bare metal* dostępny na architektury i386, amd64, ia64 oraz powerpc. Początkowo zaprojektowany i rozwijany na *University of Cambridge Computer Laboratory*, aktualnie jest własnością firmy *Citrix* ale nadal jest rozwijany przez ludzi z całego świata na otwartej licencji *GPL*. Pozwala na współdziałanie wielu systemów guest jednocześnie, zapewniając przy tym bardzo dużą wydajność i swobodę w zarządzaniu zasobami. *Xen* to narzędzia klasy *enterprise*, bez dyskusji jeden ze standardów dzisiejszych rozwiązań wirtualizacji.

W przypadku monitora *Xen* zarządzanie odbywa się przez uprzywilejowaną domenę zarządzającą *dom0*, która przez odpowiednie podsystemy monitora kontroluje zasobami pozostałych systemów guest, zwanych tutaj *domU*, warto też wspomnieć, że *dom0* posiada pełny dostęp do sprzętu, podczas, gdy domeny *domU* mają przeważnie przydzielane wirtualne zasoby, chociaż z odpowiednimi rozszerzeniami sprzętowymi, możliwe jest również im przekazanie bezpośredniej kontroli nad poszczególnymi częściami komputera.

Aktualnie monitor *Xen* jest przeportowany na trzy systemy operacyjne, które mogą pełnić rolę zarządzającą *dom0*, mianowicie Linux, NetBSD oraz OpenSolaris. O wiele więcej systemów jest przystosowanych do pracy w parawirtualizacji jako *domU* (zamiast *syscall* używa się specjalnych odwołań hypervisora czyli *hypercall*), są to między innymi Minix, Plan 9, OpenBSD, FreeBSD, NetWare czy mikrojądra Hurd oraz Mach. Oczywiście systemy, które zostały przepor-

towane na *dom0* zostały także przeportowane na *domU*. Możliwe jest też uruchamianie niezmodyfikowanych systemów operacyjnych jako domenę *HVM*, należy jednak posiadać procesor z odpowiednimi rozszerzeniami sprzętowymi jak *AMD-V* czy *VT-x*.

Wydajność jest na najwyższym poziomie, w przypadku parawirtualizacji jest ona bardzo bliska natywnej, do tego zaimplementowano specjalny mechanizm z wirtualnymi urządzeniami i parawirtualnymi sterownikami dla systemów guest, tak aby jak najwydajniej obsługiwać sieć czy operacje I/O. W przypadku trybu *HVM* wiele zależy od procesora oraz chipsetu, oraz odpowiednich rozszerzeń sprzętowych jak *VT-d*, poza tym nowsze implementacje zarówno *VT-x* jak i *AMD-V* zapewniają szybsze działanie i mniejsze opóźnienia w stosunku do poprzednich wersji.

Należy także wspomnieć, że sporo kodu, zwłaszcza związanego z wirtualizacją I/O pochodzi z projektu emulatora/monitora *QEMU*. Sam Xen składa się z mniej niż 150 000 linii kodu, co sprawia, że nie posiada żadnego zbędnego balastu opóźniającego jego działanie, a jedyne to co niezbędne. Używa istniejących już otwartych sterowników z systemu Linux, dzięki czemu nie ma potrzeby pisania ich na nowo, a poza tym dostajemy "gratis" całą masę testerów, którzy już dawno przetestowali owe sterowniki na systemie Linux.

Xen pozwala na migrację systemów guest bez zatrzymywania ich pracy, po sieci LAN pomiędzy różnymi komputerami. Pamięć systemu guest jest na bieżąco kopiowana na drugi serwer co pozwala na przeniesienie bez przerywania pracy. Potrzebna jest jedynie "przerwa" długości 60-300ms na ostateczną synchronizację i zakończenie procesu migracji

Hypervisor *xVM Server* obsługuje aktualnie następujące technologie: *pełna wirtualizacja*, *parawirtualne sterowniki*, *smp guest*, *skalowanie procesora*, *obsługa nested page table*.

2.1.2 xVM



<http://xvmserver.org>

Korporacja *Sun* wzięła ogólnie dostępny kod źródłowy projektu *Xen* i stworzyła swój produkt pod nazwą *xVM*, działa on pod kontrolą flagowego systemu operacyjnego tejże firmy, czyli *OpenSolaris*. Aktualna implementacja monitora *xVM* odpowiada wersji 3.1 monitora *Xen*.

Generalnie jego funkcjonalność odpowiada funkcjonalności monitora Xen. Przykładowy wynik polecenia `xm list` pokazującego aktualnie działające domeny przedstawia **rysunek 2.1**.

```
# xm list
Name           ID    Mem VCPUs    State    Time(s)
Domain-0       0    4947    4    r----- 151429.5
bsd70          45    1023    1    -b----- 121.3
linux26        33    1032    1    -b----- 1393.9
```

Rysunek 2.1: Wynik polecenia `xm list`.

Hypervisor *xVM Server* obsługuje aktualnie następujące technologie: *pełna wirtualizacja*, *parawirtualne sterowniki*, *smp guest*, *skalowanie procesora*, *obsługa nested page table*.

2.1.3 VMware ESX



Komercyjne produkty korporacji *VMware* stanowią największą konkurencję dla wolnych rozwiązań na rynku wirtualizacji, a monitor *ESX* jest w szczególności konkurencją dla *Xen*. Używa zmodyfikowanego systemu Linux w starszej wersji 2.4 oraz swojego jądra własnościowego¹ jądra *vmkernel*. Jadro VMware bezpośrednio zarządza pamięcią i procesorami używając techniki *scan before execution* (czyli w skrócie *SBE*, co umożliwia mu przechwytywanie i obsługę wrażliwych lub niebezpiecznych instrukcji systemów guest. Oferuje również specjalną konsolę do zdalnego zarządzania zwana *vmnix*, przedstawia ją **rysunek 2.2**.

Wiele z modułów do obsługi sprzętu monitora ESX pochodzi ze sterowników systemu Linux, a sam monitor ma zaimplementowany specjalny moduł z API systemu Linux, aby używać jego modułów bez modyfikacji. Moduły te są używane głównie do niektórych kart sieciowych, oraz interfejsu i kontrolerów SCSI. Podobnie jak w przypadku Xen, również tutaj dostarczane są specjalne parawirtualne sterowniki aby korzystać z wirtualnych urządzeń oraz specjalnych odwołań hypervisora aby maksymalnie zwiększyć przepustowość zarówno sieci jak i operacji I/O.

Pozwala na migrację w czasie rzeczywistym na inny system host, czyli *live migration* praktycznie bez zatrzymywania pracy systemu guest, podobnie jak w przypadku swojego największego

¹*proprietary*



Rysunek 2.2: Konsola zarządzająca monitorem *VMware ESX*.

szego konkurenta zapewnia wydajność na przyzwoitym poziomie. Nie ma systemów przystosowanych do parawirtualizacji na monitorze ESX, przez co wszystkie systemy działają w trybie pełnej emulacji dzięki mechanizmowi *binary translation*, należy więc spodziewać się wydajności nieco mniejszej niż natywna na tym samym sprzęcie. Wyjątkiem jest interfejs *VMI*, który jest zaimplementowany w systemach Linux, więc chociaż ten system może pracować z prędkością bliską natywnej za pomocą parawirtualizacji.

Hypervisor *VMware ESX* obsługuje aktualnie następujące technologie: *pełna wirtualizacja*, *parawirtualne sterowniki*, *smp guest*, *obsługa nested page table*, *binary translation*, *dynamic recompilation*, *direct execution*.

2.2 Hypervisor typu 2

2.2.1 QEMU



Projekt *QEMU* to projekt bardzo niedoceniany. Jest zarówno emulatorem jak i maszyną wirtualną, zależnie od tego, czy korzystamy z dodatkowego modułu jądra *kqemu* (istnieje też moduł *qvm86* ale jego rozwój został porzucony na początku 2007 roku). Autorem kodu zarówno emulatora jak i modułu jest *Fabrice Bellard*. Kod *QEMU* jest używany w praktycznie każdym projekcie związanym z wirtualizacją, zarówno otwartym jak i zamkniętym, z Xen, Win4Lin, Win4BSD, Win4Solaris, KVM i VirtualBox na czele. Cały kod źródłowy zarówno emulatora, jak owego modułu jest wolny i dostępny na wolnych licencjach *GPL*, *LGPL* oraz *BSD*

w przypadku emulacji urządzeń.

Aby zapewnić w miarę dobrą wydajność używa mechanizmów *binary translation* oraz *dynamic recompilation*, a także *direct execution* (w przypadku użycia modułu *kgemu*²). Rozwiązania te nie są jednak tak "dopieszczane" jak na przykład w projekcie VirtualBox. W projekcie używa się właśnie tych technik, aby jego kod był jak najbardziej przenośny, to tak naprawdę jego największa zaleta, może zostać bardzo łatwo przeportowany na inny system operacyjny. W trybie emulacji nie potrzebuje praw administratora do sprawnego działania. Aby wykonywać kod aplikacji (czyli *userspace*) projekt QEMU wykorzystuje kod projektów WINE oraz DOSEMU.

Używa mechanizmu *copy on write* przy zapisach danych na wirtualnych dyskach, do tego wirtualne dyski zajmują tylko tyle miejsca, ile danych się na nim znajduje, a nie na sztywno tyle ile ma rozmiar dysku. Domyślnie QEMU uruchamia się z linii poleceń, jednak powstały także graficzne nakładki pokroju QEMU Lanucher czy Qemulator.

Emulator QEMU obsługuje aktualnie następujące technologie: *binary translation*, *dynamic recompilation*, *direct execution*, *obsługa snapshot/commit*, *obsługa protokołu RDP*, *smp guest*, *przekazywanie urządzeń USB*.

2.2.2 VirtualBox



<http://virtualbox.org>

Firma Sun wykupiła na początku 2008 roku firmę Innotek, czyli twórcę oryginalnego hypervisora *VirtualBox*, od tego czasu każe nazywać swój produkt *xVM VirtualBox*. Nazwa ta jest jednak myląca ponieważ sugeruje, że VirtualBox może być częścią hypervisora xVM Server, nic bardziej mylnego, co więcej, xVM i VirtualBox nie mogą działać na raz na jednym komputerze³, ale od czego jest marketing. Firma Sun rozumie to w ten sposób, że deweloperzy mogą przetestować aplikacje w wirtualnych środowiskach na swoim systemie operacyjnym zanim zaczną używać xVM Server. Podobnie jak większość projektów *open source* związanych

²Moduł *kgemu* obsługuje aktualnie architektury *i386* oraz *amd64*.

³Hypervisor typu 1 nie może współdziałać na jednym komputerze z hypervisorem typu 2.

z wirtualizacją również korzysta z kodu źródłowego emulatora/hypervisora *QEMU* i jego rozwiązań, jak *dynamic recompilation* gdy nie można użyć innych mechanizmów.

VirtualBox to klasyczny *hosted hypervisor*, aktualnie jeden z najwydajniejszych. Rozwijany jest na otwartej licencji *GPL2*. Systemem host mogą być systemy Linux, Windows, OpenSolaris oraz Mac OS X. Możliwy jest dosyć "łatwy" port na system FreeBSD, jednak brakuje dewelopera, który przeportowałby moduł jądra na tenże system. Wirtualizacja odbywa się tutaj za pomocą technologii *binary translation* oraz *direct execution* ale możliwe jest także korzystanie ze sprzętowych rozszerzeń wirtualizacji. System guest może być dowolny. VirtualBox oferuje wirtualizację na architekturach i386 oraz amd64.

Po tym jak *Sun* przejął twórcę VirtualBox, zintegrował ten hypervisor tak dobrze jak tylko mógł ze swoim wiodącym systemem operacyjnym, czyli OpenSolaris, jednocześnie wspierając porty na pozostałe systemy operacyjne. Dzięki tej bliskiej więzi VirtualBox współpracuje z *Solaris Containers* oraz aktualnie w fazie testowej rozwiązaniem *Crossbow*, czyli wirtualizacją stosu sieciowego.

VirtualBox to rozwiązanie najlepiej nadające się na stacje robocze oraz desktop, zwłaszcza ze swoimi funkcjami integracji z pulpitem systemu host, czyli obsługę trybu *seamless mode*. Obsługuje także protokół *RDP*⁴. Posiada także wiele innych rozwiązań, obsługę standardu *iSCSI*, kolejkowania rozkazów *NCQ* w przypadku używania partycji czy całego dysku SATA bezpośrednio. Pozwala na przekazywanie urządzeń *USB* bezpośrednio do systemu guest, a także wiele innych usprawnień w komunikacji z systemie host. Domyślnie *VirtualBox* używa emulowanej karty graficznej z 8MB pamięci ale ze specjalnymi rozszerzeniami *Guest Additions* pozwala na o wiele lepszą wydajność generowania grafiki łącznie z dynamiczną zmianą rozdzielczości w systemie guest, gdy po prostu zmieniamy rozmiar okna.

Jest jednak pewna "niedogodność", niektóre z wymienionych funkcjonalności jak przekazywanie urządzeń *USB* oraz obsługa interfejsu *iSCSI* są dostępne w pełnej wersji VirtualBox, która jest dostępna jedynie w binarnej formie na innej licencji, które pozwala na darmowe użycie owego oprogramowania tylko na własny użytek. Również kod źródłowy tych rozwiązań nie został opublikowany.

Hypervisor *VirtualBox* obsługuje aktualnie następujące technologie: *seamless mode*, pełna wirtualizacja, parawirtualne sterowniki, *smp guest*, skalowanie procesora, obsługa *nested page table*, przekazywanie urządzeń *USB* i *SCSI*, obsługa *NCQ*, obsługa protokołu *RDP*, *binary translation*, *dynamic recompilation*, *direct execution*.

⁴Remote Desktop Protocol

2.2.3 KVM



<http://linux-kvm.com>

Firma Qmuranet stworzyła maszynę pod nazwą *Kernel based Virtual Machine*, co daje nam w skrócie *KVM*, jest ona silnie związana z jądrem systemu Linux, a od jego wersji 2.6.20 jest nawet jego częścią. Został również przeportowany na system FreeBSD w 2007 roku, jednak od tamtego czasu port ten nie był rozwijany, co oznacza, że przez zmiany w projekcie po prostu już nie działa. Aktualnie możliwa jest wirtualizacja tylko z asystą sprzętowych rozszerzeń jak *AMD-V* oraz *VT-x*. Podobnie jak kilka poprzednich projektów, również tutaj zaimplementowano odpowiednią infrastrukturę dla parawirtualnych sterowników urządzeń w systemach guest. Aktualnie działa w formie modułu jądra, wykorzystując dużą część kodu projektu *QEMU* dla emulacji urządzeń, wyświetlania obrazu i prawie wszystkich innych aspektów, oprócz samej wirtualizacji procesora.

Cały kod dostępny jest na otwartej licencji *GPL* co z resztą jest raczej oczywiste, gdyż licencja ta nie pozwala na linkowanie kodu z kodem na jakiegokolwiek innej licencji⁵, a na *GPL* jest przecież jądro Linux. Aktualnie obsługuje architektury *i386* oraz *amd64* (aktualnie trwają prace nad portami na architektury *powerpc* oraz *ia64*). Dzięki temu, że jest tylko modułem jądra Linux, korzysta z wszystkich jego funkcji, jak na przykład skalowanie procesora, czy używanie pamięci *swap* systemu host.

Sama wydajność rozwiązania KVM jest bardzo dobra, ale jej wydajność zależy także od tego jak szybko działają sprzętowe rozszerzenia w procesorze. Aplikacja *Virtual Machine Manager* pozwala na graficzne tworzenie i zarządzanie wirtualnymi maszynami korzystającymi z KVM

Monitor KVM obsługuje aktualnie następujące technologie: *pełna wirtualizacja, parawirtualne sterowniki, migracja bez przerywania pracy, smp guest, skalowanie procesora, obsługa snapshot/commit, obsługa nested page table, obsługa protokołu RDP, przekazywanie urządzeń USB i SCSI*.

2.2.4 VMware Server



<http://vmware.com/products/server>

⁵Jest to uznawane zarówno za jej największą zaletę jak i wadę jednocześnie.

Hypervisor *VMware Server* (kontynuator starej linii produktów pod nazwą *GSX Server*) to najprostszy produkt firmy *VMware* przeznaczony do wirtualizacji serwerów. Systemem host może być Linux lub Windows, system guest jest dowolny, gdyż wirtualizacja odbywa się przez rozwiązania takie jak *binary translation*, czy *direct execution*. Działa na architekturach *i386* oraz *amd64*.

Wydajność stoi na dobrym poziomie, choć należy pamiętać, że produkt ten jest przeznaczony jedynie na rynek serwerów i został pozbawiony jakichkolwiek rozwiązań związanych z pracą na stacjach roboczych czy desktopach. *VMware Server* obsługuje interfejs parawirtualizacji *VMI*, oraz pozwala zarządzać maszynami wirtualnymi przez interfejs przeglądarki stron *www*, czy też specjalną konsolę zdalną.

Pomimo swoich zalet bywa czasem problematyczny, gdyż "pochłania" przerwania procesora, co utrudnia poprawne utrzymywanie aktualnego czasu, sprawia to oczywiście problemy z usługami i serwerami, które polegają na ścisłych zmianach w czasie. Aby zminimalizować tę niedogodność co jakiś czas systemów guest są synchronizowane z czasem systemu host. Można oczywiście zawsze skorzystać z serwera czasu na systemie host, do którego będą się odwoływać systemy guest.

Hypervisor *VMware Server* obsługuje aktualnie następujące technologie: *pełna wirtualizacja*, *parawirtualny interfejs VMI*, *smp guest*, *binary translation*, *direct execution*.

2.2.5 VMware Workstation



Hypervisor *VMware Workstation* firm *VMware* jest z kolei produktem przeznaczonym do pracy na stacjach roboczych oraz desktopach. Systemem host może być Linux lub Windows, system guest może być dowolny. Pozwala na wirtualizację na architekturach *i386* oraz *amd64*. Używa tradycyjnych dla *VMware* technik jak *binary translation* czy *direct execution*. W przeciwieństwie do *VMware Server* posiada wiele usprawnień jak choćby grafika 3D w systemach guest dzięki pełnej obsłudze API Direct3D 8 oraz częściowej obsłudze API Direct3D 9.

Produkt *VMware Workstation* gnębią te same problemy co wersje serwerową czyli problemy z synchronizacją czasu w systemach guest. Posiada za to ciekawą funkcję *Shared Folders*, dzięki której mamy dostęp z systemu guest do plików na systemie host. Zastosowania tego produktu są dodatkowo wymuszane przez jego restrykcyjną licencję, która nie pozwala na uruchamianie na owym hypervisorze serwerów.

Hypervisor *VMware Workstation* obsługuje aktualnie następujące technologie: *pełna wirtualizacja, binary translation, direct execution, grafika 3D, współdzielenie plików*.

2.2.6 VMware Fusion



<http://vmware.com/products/fusion>

Ostatnim z rodziny produktów firmy *VMware* jest *VMware Fusion*. Hypervisor ten przeznaczony jest jedynie na system operacyjny Mac OS X (oraz tylko na procesorach Intel). Używa takich rozwiązań jak *binary translation* czy *direct execution*. Dzięki implementacji trybu *seamless mode*⁶ zapewnia dużą wygodę w działaniu z systemem guest. Pozwala też na SMP w systemach guest.

Obsługuje także grafikę 3D w systemach guest, API DirectX 9.0 oraz OpenGL przez mechanizm *direct recompilation*. Posiada rozwiązania takie jak *snapshot/commit* czy współdzielenie plików pomiędzy systemem guest i host. Potrafi także przekazywać urządzenia USB bezpośrednio do systemu host. Innym ciekawym rozwiązaniem jest mapowanie skrótów klawiaturowych pomiędzy systemem guest i host co pozwala na bardziej produktywną pracę.

Hypervisor *VMware Fusion* obsługuje aktualnie następujące technologie: *seamless mode, pełna wirtualizacja, smp guest, przekazywanie urządzeń USB, binary translation, dynamic recompilation, direct execution, grafika 3D, współdzielenie plików*.

2.2.7 Parallels Desktop



<http://parallels.com/products/desktop>

Hypervisor *Parallels Desktop* przeznaczony jest na system Mac OS X. Używa standardowych technologii dla hypervisorów typu 2, czyli *binary translation, dynamic recompilation* oraz *direct execution*. Pozwala na przekazywanie urządzeń *USB* do systemów guest, a także tryb *seamless mode*, zwany tutaj *coherence*. Posiada także implementację dzielonego schowka (czyli *clipboard*) a także mechanizmu *drag and drop* pomiędzy systemami guest oraz host. Jak widać jest to hy-

⁶Tryb ten zwany jest tutaj *Unity*.

pervisor przeznaczony *stricto* na stację roboczą lub po prostu desktop.

Obsługuje także grafikę 3D w systemach guest, najprawdopodobniej przez implementację API systemu Windows przez kod biblioteki projektu *WINE*⁷, który już od ponad 15 lat rozwija owe API na otwartej licencji *LGPL*. Jak wiadomo produkt firmy *Parallels* jest aplikacją zamkniętą, ale zgodnie z licencją *WINE* firma musi "oddać" w postaci kodu źródłowego wszystkie zmiany jakie tam wprowadziła, odbyło się to z pewnym opóźnieniem, jednak firma wywiązała się ze swych obowiązków.

Aktualnie *Parallels Desktop* zapewnia obsługę API DirectX 8.1 oraz OpenGL. Posiada również technologię *SmartSelect*, która pozwala wybrać czy dany plik otworzymy aplikacją systemu host, czy też aplikacją systemu guest, zapewnia także współdzielenie plików pomiędzy systemami.

Hypervisor *Parallels Desktop* obsługuje aktualnie następujące technologie: *seamless mode*, pełna wirtualizacja, *smp guest*, przekazywanie urządzeń USB, *binary translation*, *dynamic recompilation*, *direct execution*, grafika 3D, współdzielenie plików.

2.2.8 Parallels Workstation



<http://parallels.com/products/workstation>

Produkt *Parallels Workstation* przeznaczony jest zarówno na system Mac OS X (pod nazwą *Parallels Workstation for Mac*, jak i na systemy Linux oraz Windows. Podobnie jak inne produkty z rodziny firmy *Parallels* używa rozwiązań takich jak *binary translation*, *dynamic recompilation* oraz *direct execution*. Obsługuje także sprzętowe rozszerzenia wirtualizacji *AMD-V* oraz *TV-x*. Zapewnia przekazywanie urządzeń USB do systemów guest, nie pozwala jednak na *SMP*. Mimo iż jest to produkt przeznaczony na stacje robocze oraz desktop, to brakuje mu sporej części funkcjonalności z *Parallels Desktop*.

Hypervisor *Parallels Workstation* obsługuje aktualnie następujące technologie: pełna wirtualizacja, przekazywanie urządzeń USB, *binary translation*, *dynamic recompilation*, *direct execution*.

2.2.9 Parallels Server

Hypervisor *Parallels Server* przeznaczony jest głównie na system operacyjny Mac OS X (gdzie nosi nazwę *Parallels Server for Mac*. Na systemy Linux oraz Windows dostępna jest

⁷*WINE* znaczy dokładnie *Wine Is Not an Emulator*.



<http://parallels.com/products/server>

aktualnie wersja *beta*. Używa rozwiązań takich jak *binary translation*, *dynamic recompilation* oraz *direct execution*. Obsługuje także sprzętowe rozszerzenia wirtualizacji AMD-V oraz TV-x. Pozwala na na SMP w systemach guest.

Tak naprawdę produkt ten różni się od poprzednich głównie nazwą oraz licencją pod którą jest sprzedawany, gdyż rozwiązania są bardzo podobne, a często nawet po prostu takie same. Oferuje jednak specjalną konsolę do zarządzania maszynami wirtualnymi oraz narzędzia do migracji systemów guest.

Hypervisor *Parallels Server* obsługuje aktualnie następujące technologie: *pełna wirtualizacja*, *binary translation*, *dynamic recompilation*, *direct execution*, *smp guest*.

2.2.10 Win4Lin / Win4BSD / Win4Solaris



<http://win4lin.net> <http://win4bsd.com> <http://win4solaris.com>

Win4Lin to zamknięty hypervisor firmy która aktualnie przyjęła nazwę *Virtual Bridges*⁸, przeznaczony jest do wirtualizacji systemów Windows na systemie Linux. Wiele kodu pochodzi z projektu QEMU (łącznie z kodem modułu *kqemu*).

Zastosowane technologie nie odbiegają znacznie od tych zastosowanych w QEMU, czyli *binary translation*, *direct execution* czy *dynamic recompilation*, choć pewnie sporo zmodyfikowane pod kątem wydajności, ale za to znacznie mniej przenaszalne niż oryginalny kod projektu QEMU. Posiada też ciekawą funkcję współdzielenia plików pomiędzy systemami host i guest, mianowicie, na pulpicie systemu guest jest specjalny katalog, który zapewnia bezpośredni dostęp do plików katalogu domowego użytkownika z prawem zapisu i odczytu.

Produkt ten miał duże znaczenie w czasach systemów Windows z rodziny 9x, gdy otwarte rozwiązania praktycznie nie istniały, do tego *Win4Lin* zapewniał praktycznie natywną wydajność dla starych systemów Windows. Aktualnie brakuje mu dzisiejszych rozwiązań aby skutecznie konkurować z innymi popularnymi, a przede wszystkim domowymi rozwiązaniami, bo przecież za każdą licencję *Win4Lin* należy zapłacić.

⁸W przeszłości istniała także pod nazwami *Win4Lin* oraz *Netraverse*

Istnieją także odmiany dla systemów FreeBSD (pod nazwą *Win4BSD*) oraz Solaris (pod nazwą *Win4Solaris*). Obsługują te same technologie oraz zapewniają taką samą funkcjonalność, różnica tkwi w innej architekturze modułu jądra hypervisora, który musiał po prostu zostać przeportowany na jądro innego systemu operacyjnego. Warto jednak zaznaczyć, że wersja *Win4BSD* jest darmowa dla użytkownika domowego.

Hypervisor *Win4Lin* obsługuje aktualnie następujące technologie: *pełna wirtualizacja, współdzielenie plików, binary translation, dynamic recompilation, direct execution*.

2.3 Na poziomie systemu operacyjnego

2.3.1 FreeBSD Jails



<http://freebsd.org>

Mechanizm *FreeBSD Jails* jest częścią samego systemu operacyjnego i możemy go uaktywnić w dowolnej chwili. Pozwala bezpiecznie odseparować każdą wirtualną maszynę zarówno od pozostałych maszyn, jak i od systemu host. Każdy *Jail* może posiadać swój adres IP i odpowiednio zmodyfikowany system FreeBSD w swoim katalogu. Narzut związany z wirtualizacją jest praktycznie zerowy (można także powiedzieć że po prostu go nie ma), gdyż każda maszyna korzysta i komunikuje się bezpośrednio z jądrem systemu host, bez typowych "spowalniaaczy" pokroju podwójnego kopiowania, czy też specjalnych odwołań. Do tego nie musimy na sztywno ustawiać rozmiaru pamięci dla każdej maszyny, gdyż jest dynamicznie przydzielana w miarę potrzeby. Rozwiązanie to, podobnie jak cały system FreeBSD rozwijane jest na licencji *BSD*.

Ułatwia także administrację, gdyż w każdej chwili możemy podmienić jedną z naszych usług na inny jej wariant, czy to inaczej skonfigurowany, czy też na nowszą wersję, a w razie jakichkolwiek problemów po prostu przywrócić starszą wersję. Każda z instancji posiada swoich użytkowników, swoje procesy, również odseparowane od procesów systemu host (tak zwany *sandbox*) co jest bardzo ważne przy problemach typu *buffer overflow* (czyli po prostu przepełnienie bufora). Zapewnia to bardzo dobre bezpieczeństwo. Dodatkowo system FreeBSD oferuje specjalny mechanizm automatycznego wykrywania zarówno *buffer over-*

flow jak i *buffer underflow* (niedopełnienie bufora), co zapewnia dodatkowe bezpieczeństwo. Dodatkowo możemy skorzystać z mechanizmu FreeBSD o nazwie *securelevel* (zarówno dla systemu host jak i każdego z systemów guest). Rozwiązanie to na najwyższym stopniu bezpieczeństwa odbiera uprawnienia nawet najważniejszemu użytkownikowi na systemach UNIX czyli użytkownikowi *root*, nawet więc jak ktoś się włamie to nic nie zrobi bo system mu na to nie pozwoli.

Sam mechanizm *Jail* jest po prostu bardzo rozszerzoną wersją polecenia *chroot* z systemów UNIX, samo polecenie *chroot* nie izoluje jednak przestrzeni adresowej procesów, ani nie zapewnia żadnej z funkcjonalności jakie oferuje mechanizm systemu FreeBSD. Dodatkowo możemy bezpiecznie udostępnić zasoby systemu host w systemie guest za pomocą systemu plików *nullfs* z systemu FreeBSD, co w wielu przypadkach na pewno zaoszczędzi nam sporo miejsca.

Dzięki mechanizmowi *jail* tak naprawdę możemy uruchomić tyle wirtualnych systemów ile tylko nam się podoba, nie ma żadnego limitu, który nas ogranicza, co najwyżej przestrzeń dyskowa oraz wolna pamięć. Nowy system *Jail* zajmuje trochę ponad 100MB, jednak stosując metodę *ezjail*, która zamiast kopiować te same aplikacje do każdego z wirtualnych systemów po prostu odwołuje się do poleceń systemu host w trybie read only, podobnie z innymi zasobami, które występują w każdym z takich systemów. Ogranicza to rozmiar każdej nowej maszyny do około 2MB, co sprawia, że miejsce na dysku przestaje mieć tutaj jakiegokolwiek znaczenie i musimy tylko zadbać o odpowiednią ilość pamięci.

FreeBSD posiada także technologię, która pozwala na uruchamianie niezmodyfikowanych aplikacji binarnych z systemu Linux, nosi ona nazwę *Linux binary compatibility*. Nic nie stoi więc na przeszkodzie, aby w środowisku *Jail* używać także aplikacji z systemu Linux dostępnych tylko w formie binarnej.

2.3.2 Solaris Containers



<http://sun.com/software/solaris/containers>

Technologia *Solaris Containers* (łącznie z *Solaris Zones*) jest częścią systemu operacyjnego Solaris (także OpenSolaris). Pozwala tworzyć maszyny wirtualne odseparowane od systemu host, posiadające własnych użytkowników, własne usługi i procesy, ale z izolacją od procesów systemu host. Każda taka instancja może być dowolnie zmodyfikowaną wersją systemu Solaris (również starszych wersji tego systemu). Wydajność takiego rozwiązania jest tożsama

z wydajnością natywną, gdyż każda z takich maszyn korzysta bezpośrednio z jądra systemu host, za pomocą standardowych odwołań tegoż systemu, czyli *syscall*.

Mamy możliwość przydzielania na sztywno określonej ilości procesorów oraz pamięci dla każdej maszyny. Możemy też pozostawić przydzielanie zasobów systemowi Solaris. Możemy też przydzielić całe pule zasobów odpowiednim grupom naszych maszyn, system Solaris dostarcza tutaj bardzo dużą elastyczność w zarządzaniu zasobami. System host nazywany jest tutaj jako *global zone*, podczas gdy systemy guest nazywane są po prostu *zones*. Dla każdej takiej maszyny możemy także przydzielić adres IP, tak jak dla zwykłego komputera. Na jednym systemie Solaris możemy uruchomić maksymalnie 8191 instancji systemów guest, 8192 licząc z systemem host.

Tak zwane *sparse zones* pozwalają na współdzielenie wspólnych zasobów z systemem host, przez co instancja taka zajmuje tylko 50MB, podczas gdy instancja, która posiada pełny system Solaris będzie zajmowała kilkaset megabajtów, w skrajnych sytuacjach nawet do kilku gigabajtów. Ograniczeniem *Containers* jest na przykład brak możliwości korzystania z dzielenia zasobów protokołem *NFS*, może to robić jedynie system host.

Istnieje też możliwość używania tak zwanych *branded zones* (w skrócie *BrandZ*), czyli instancji, które nie korzystają z jądra systemu host. Aktualnie możliwe jest używanie jąder systemów Solaris 8, Solaris 9 oraz jądra systemu Linux. Dodatkowo rozszerza to możliwości tego rozwiązania, zwłaszcza możliwość używania jądra Linux. Do tego technologie takie jak system plików *ZFS* czy *DTrace* do wyjątkowo dokładnego tuningowania wydajności aplikacji jak i samego systemu zapewniają naprawdę bogate środowisko do pracy.

2.3.3 Linux VServer



Technologia *Linux VServer* pozwala stworzyć wiele współdziałających maszyn wirtualnych korzystających bezpośrednio z jądra systemu Linux. Wymaga jednak nałożenia odpowiednich nakładek (czyli po prostu *patch*) na jądro systemu host. Zapewnia natywną wydajność dzięki używaniu odwołań *syscall*. Maszyny wirtualne są od siebie odizolowane, każdy z nich posiada osobną bazę użytkowników, procesów, strukturę katalogów ale własnego odrębnego adresu IP już sobie nie przydzielimy, gdyż *VServer* nie zapewnia wirtualizacji stosu sieciowego.

Istnieje także projekt *Linux Virtual Server*, który zapewnia balansowanie ruchu sieciowego, ale nie należy w żaden sposób łączyć tych dwóch projektów gdyż nie mają ze sobą nic wspól-

nego poza podobną nazwą. Przestrzeń adresowa procesów każdego z systemów guest jest odseparowana od pozostałych instancji (tak zwany *sandbox*), co zapewnia duże bezpieczeństwo takiego rozwiązania. Rozwiązanie *Linux VServer* dostępne jest na licencji *GPL*.

Systemy guest dzielą pomiędzy siebie dostępne zasoby komputera ale nic nie stoi na przeszkodzie, aby konkretnej maszynie przydzielić kilka dedykowanych procesorów oraz większą ilość pamięci. Osobne drzewo plików i katalogów takiej wirtualnej instancji zajmuje mało miejsca przez współdzielenie tak wielu rzeczy z systemem host jak to tylko możliwe na zasadzie *copy on write*. Z wad tego rozwiązania można wspomnieć nie tyle wadę samego rozwiązania, co wybór jądra Linux, które nie jest tak bezpieczne jak jądra systemów FreeBSD czy Solaris.

2.3.4 Linux OpenVZ



Rozwiązanie *Linux OpenVZ* jest konkurencją dla rozwiązania *Linux VServer*, również wykorzystuje jądro systemu Linux aby serwować wirtualne instancje na nim oparte. Można powiedzieć, że wydajność również jest taka sama, ponieważ podobnie jak konkurent korzysta z odwołań *syscall*. Wymaga oczywiście nałożenia odpowiednich nakładek *patch*. Rozwiązanie to jest dostępne na otwartej licencji *GPL*. Projekt ten jest także bazą dla innego rozwiązania, konkretnie dla *Parallels Virtuozzo Containers*, zamkniętego rozwiązania firmy *Parallels*. Warto o tym wspomnieć, gdyż *OpenVZ* otrzymuje wsparcie finansowe od tej właśnie firmy.

Każda z wirtualnych instancji posiada swoje drzewo katalogów, użytkowników, adres IP, wirtualizowane systemy plików */proc* oraz */sys*. Jest to wbrew pozorom bardzo duża zaleta gdyż *Linux VServer* nie oferuje takiego rozwiązania, a jądro systemu Linux wymaga tych systemów plików do poprawnego działania. Oczywiście nie zapomniano o takich podstawach jak izolacja przestrzeni adresowej procesów oraz wirtualizacji stosu sieciowego. Dodatkowym atutem jest możliwość przypisania danej instancji karty sieciowej czy też portu *serial*.

Zasoby są współdzielone pomiędzy działające instancje, w podobny sposób jak ma to miejsce przy zwykłych procesach systemu, możliwe jest jednak dowolne priorytetowanie poszczególnych instancji, a także przydzielanie na sztywno czasu procesora. Podobnie ma się sprawa z przypadkiem operacji I/O. Inną bardzo ważną zaletą rozwiązania *OpenVZ* jest migracja pomiędzy fizycznymi serwerami, wymaga jednak zamrożenia danej instancji na czas jej przeniesienia, co trwa jednak zwykle nie więcej niż kilka do kilkunastu sekund. Nie ma żadnego odgórnego limitu uruchomionych instancji, ograniczają nas tylko wolne zasoby komputera.

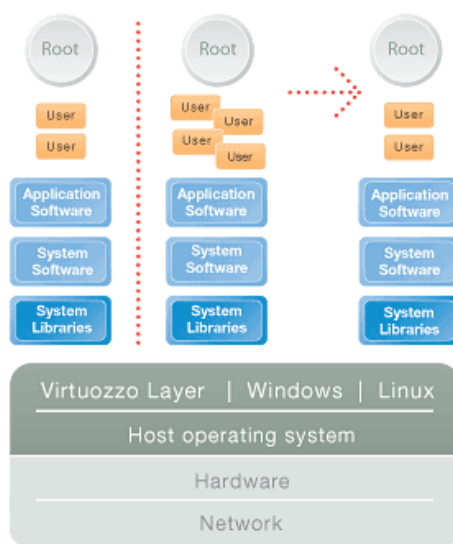
Na koniec może małe zestawienie *OpenVZ* ze konkurentem, czyli z *VServer*. *OpenVZ* jest mówiąc bardzo ogólnie rozwiązaniem dojrzalszym i bardziej elastycznym, zapewniając wydajność na tym samym poziomie co konkurent. Posiada o wiele więcej możliwości zarządzania, a przez wirtualizację stosu sieciowego oraz wirtualnych systemów plików zapewnia, że wszystkie aplikacje będą działać w wirtualizowanych instancjach prawidłowo.

2.3.5 Parallels Virtuozzo Containers



<http://parallels.com/virtuozzo>

Rozwiązanie *Parallels Virtuozzo Containers* firmy *Parallels* bazuje na technologii *OpenVZ*. Występuje w dwóch wersjach, na system Linux oraz na system Windows. W pierwszym przypadku jest to po prostu funkcjonalność rozwiązania *OpenVZ* które z resztą firma *Parallels* nie omieszcza finansować i wspierać. Wydajność jest na poziomie natywnym, oczywiście z większą ilością współdziałających maszyn wirtualnych oraz w zależności od ich obciążenia wydajność będzie odpowiednio mniejsza. Logiczny przekrój wirtualizacji jaki zapewnia rozwiązanie *Parallels Virtuozzo Containers* przedstawia **rysunek 2.3**.



Rysunek 2.3: Schemat rozwiązania *Parallels Virtuozzo Containers*.

Wersja dla systemu Windows różni się głównie tym, że narzędzia do tworzenia i zarządzania są graficzne, wszystko możemy wyklikać w okienkach co jest z resztą tradycyjnym podejściem na systemach rodziny z Redmond. W przypadku systemu Linux powita nas instalacja w tekstowym interfejsie *ncurses* dostępne jest także czysto tekstowe narzędzie, dzięki któremu

możemy wykonywać spersonalizowane nieinteraktywne instalacje.

2.3.6 User Mode Linux



<http://user-mode-linux.sourceforge.net>

Rozwiązanie *User Mode Linux* (czyli w skrócie *UML*) to specjalna wersja jądra systemu Linux, która może być wykonywana w taki sam sposób jak każdy inny proces. Oznacza to, że nie musimy posiadać praw administratora aby używać takiego wirtualnego systemu. Mimo iż żaden *patch* na jądro systemu host nie jest wymagany, to jednak zaleca się zastosowanie odpowiednich łatek w celu zwiększania wydajności rozwiązania *User Mode Linux*. Zaletą jest na pewno fakt, iż od 2004 roku *UML* jest częścią głównej gałęzi jądra Linux, a co za tym idzie jest na otwartej licencji *GPL*.

Aby system guest posiadał dostęp do sieci Internet musimy ręcznie stworzyć odpowiedni mostek *bridge*, innym wyjściem jest *routing* przez system host. Możemy także skorzystać z narzędzi do zarządzania instancjami *UML* jak na przykład *UMLazi*, znacznie ułatwi to administrację. Niestety wydajność nie stoi na tak wysokim poziomie jak w innych tego typu rozwiązaniach jak *OpenVZ* czy *VServer*.

2.3.7 Cooperative Linux



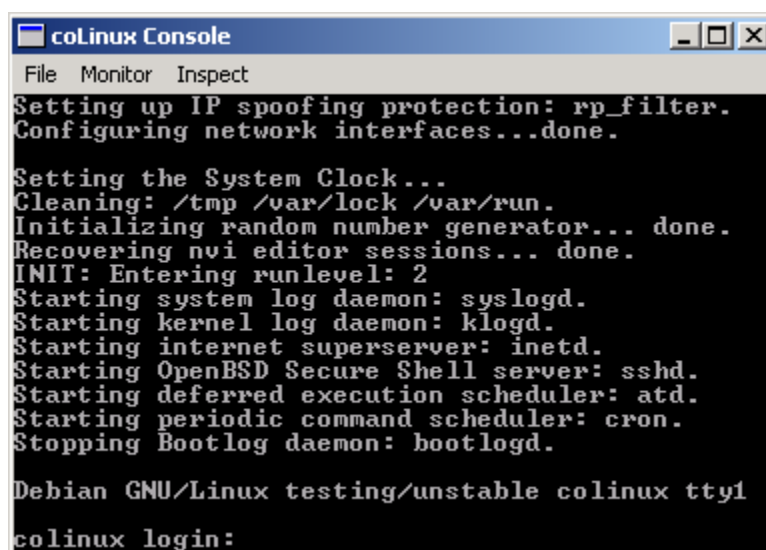
<http://colinux.org>

Rozwiązanie *Cooperative Linux* (nazywane także w skrócie *coLinux*) pozwala na uruchomienie jądra systemu Linux, na komputerze z uruchomionym już systemem Windows, czyli pozwala na współzystywanie dwóch zupełnie innych jąder na jednej maszynie w tym samym czasie. Technologia ta bazuje na koncepcji zwanej *Cooperative Virtual Machine*, która w przeciwieństwie do standardowego podejścia z systemem host i guest nie używa wirtualizowanych (lub

też emulowanych) peryferiów, tylko zapewnia obu jądom bezpośredni dostęp do fizycznego sprzętu komputera jednocześnie.

W teorii każde z jąder posiada wyłączny dla siebie kompletny kontekst procesora oraz odrębną przestrzeń adresową, a w przypadku dostępu do sprzętu, każde z jąder decyduje kiedy oddać kontrolę "przeciwnikowi" nad danym sprzętem. Architektura *i386* nie jest jednak zaprojektowana do pracy z dwoma systemami jednocześnie, co prowadzi do wielu konfliktów a nawet niestabilności gdyby obydwu jądom dać pełne uprawnienia w zarządzaniu komputerem. Wprowadzono tutaj pojęcia jądra host oraz jądra guest, gdzie jądro host kontroluje cały sprzęt, a jądro guest komunikuje się z jądrem host za pomocą specjalnego API.

Aktualnie działa na systemach Windows XP/2000 oraz Linux, ale nic nie stoi na przeszkodzie aby przeportować go na inne systemy, wszakże cały projekt jest dostępny na otwartej licencji *GPL*. Niestety nie ma nic za darmo, podejście to posiada wiele wad, jak na przykład niestabilność (po prostu zobaczymy *kernel panic*, chociaż na systemach Windows to żadna nowość tak naprawdę) oraz mniejsze bezpieczeństwo, na przykład jeżeli atakujący odpowiednio spreparuje moduł jądra *coLinux*. Rozwiązanie *coLinux* działające pod systemem Windows przedstawia **rysunek 2.4**.



Rysunek 2.4: Rozwiązanie *CoLinux* działające pod kontrolą systemu *Windows*.

Przez dość nietypową strukturę urządzeń instalowanie dystrybucji systemu Linux jest dosyć trudne dlatego przeważnie przenosi się działającą instalację systemu, lub też specjalnie przygotowany obraz do działania pod *coLinux*. Inną wadą jest brak możliwości uruchamiania systemu *X window system*, a co za tym idzie aplikacji graficznych ale aplikacje tekstowe jak serwery działają bez zarzutu.

2.4 Emulatory

2.4.1 QEMU



Projekt *QEMU* jest osobno wspomniany jako emulator, gdyż taka była jego podstawowa idea, mimo iż jest szeroko wykorzystywany jako rozwiązanie do wirtualizacji. Pozwala emulować architektury takie jak *i386*, *amd64*, *alpha*, *arm*, *powerpc*, *powerpc64*, *mips*, *sparc* oraz *sparc64*. Co więcej pozwala to robić także na nich, na przykład architekturę *arm* na architekturze *sparc64*. Dla systemu guest dostarczany jest w pełni emulowany kompletny komputer z wszystkimi peryferiami i podzespołami.

2.4.2 Bochs



Podobnie jak *QEMU* emulator *Bochs* jest bardzo przenośny, do tego pozwala emulować nie tylko cały wirtualny komputer, ale także BIOS oraz konkretne modele procesorów wraz z ich sprzętowymi rozszerzeniami jak *MMX* czy *SSE*. Dodatkowo pozwala uruchamiać aplikacje *i386* oraz *amd64* na innych architekturach, świetnie nadaje się zatem do rozwijania oraz debugowania oprogramowania to wirtualizacji. Jest dostępny na otwartej licencji *GPL*.

Jako emulator zapewnia bardzo małą wydajność, ale nie ma to znaczenia, gdyż przeznaczony jest do innych zadań niż wirtualizacja. Autorem jest Kevin Lawton lecz po uwolnieniu kodu projekt rozwijają ludzie z całego świata. Początkowo był dostępny jako w cenie \$25 jednak w 2000 roku firma *MandrakeSoft*⁹ wykupiła kod i wypuściła go na otwartej licencji *GPL*.

⁹Aktualnie *Mandriva*.

Rozdział 3

Wydajność maszyn wirtualnych

Testując różne systemy operacyjne na danym komputerze musimy zadbać o wiele czynników, o powtarzalne środowisko testowe, czy testy były wykonywane na maszynie *cold boot* (komputerze który dopiero co został włączony) czy może *hot boot* (po restarcie komputera). Czynniki pozornie nieistotne są jednak bardzo ważne w przypadku przeprowadzania dokładnych testów wydajności samych systemów operacyjnych, a w przypadku testowania rozwiązań wirtualizacji są jeszcze ważniejsze. Musimy też wiedzieć co i jak chcemy testować, gdyż wyniki niektórych testów, mimo iż poprawnie przeprowadzone, nie powiedzą nam tak naprawdę nic o wydajności danego rozwiązania.

Weźmy na przykład dość popularny test, używany zarówno przy porównaniach samych systemów operacyjnych jak i maszyn wirtualnych, mianowicie *SPECjbb* [SPEC, 2005]. Jego zadaniem jest mierzenie wydajności serwera aplikacji *Java*. Benchmark ten praktycznie nie używa czasu *kernel space*, podczas gdy typowe obciążenie tego typu mieści się w przedziale 20-30%. Hypervisor po prostu w ogóle nie ma szansy się wykazać, gdyż praktycznie przez cały czas testu pozostaje bezczynny wykonując kod aplikacji w *user space*. To tak jak postawić samochód *Porsche* w korku ulicznym i wychwalać moc jego silnika [de Gelas, 2008b]. Inną "wadą" tego testu jest brak obciążenia operacjami I/O, gdyż optymalizacje (lub ich brak) w tej kwestii są bardzo wyraźne, co pozwala łatwo oddzielić wydajny hypervisor od przereklamowanego produktu.

Istnieją także specjalne testy wydajności maszyn wirtualnych jak na przykład *VMmark* firmy *VMware*. Już na pierwszy rzut oka podejrzanie powinno być, że firma która sama jest jedną ze stron w wyścigu wydajności maszyn wirtualnych oferuje swój program do testowania ich wydajności. Co prawda jest on w części programem *open source*, jednak nie w pełni więc nie można do końca sprawdzić co i w jaki sposób sprawdza. Inną kwestią jest adnotacja w FAQ owej aplikacji, która brzmi "*VMmark is neither a capacity planning tool nor a sizing tool.*" (oprogramowanie *VMmark* nie może być używane w celu planowania "pojemności" ani "rozmiarowości"). Jaki jest więc cel używania go skoro nawet jego twórcy ostrzegają nas, że nic on nam nie powie?

Jednym z zadań wirtualizacji jest wirtualizacja systemów z rodziny Windows, dlatego by więc nie wsiąść jednego z najbardziej znanych testów, skoro wszyscy go używają do mierzenia wydajności komputerów pod działaniem systemu Windows, mowa oczywiście o aplikacji *PCMark*. Większość procesorów architektur *i386* oraz *amd64* ma na sztywno ustawiony *CPUID*, czyli unikalny identyfikator modelu procesora, co oznacza że nie można go w żaden sposób zmodyfikować (bo tak naprawdę po co). Niedawno pojawił się jednak na rynku procesor, który daje taką możliwość czyli *VIA Nano*.

Przy domyślnych ustawieniach procesor *VIA Nano* był o około 25% wolniejszy od swojego konkurenta *Intel Atom*. Po ustawieniu *CPUID* na *AuthenticAMD* (identyfikator procesorów *AMD*) nagle zyskał 10% więcej wydajności. Najciekawszy jednak jest kolejny wynik, w którym zmieniono identyfikator na *GenuineIntel* (identyfikator procesorów firmy *Intel*). Po tej zmianie procesor "*Intel*" *Nano* uzyskał wynik o prawie 50% lepszy niż pod swoją domyślną nazwą *VIA Nano*, dodatkowo wyprzedzając swojego konkurenta *Atom* ze stajni firmy *Intel* [Hruska, 2008].

Testy były przeprowadzone wiele razy, więc nie ma tu mowy o jakiegokolwiek pomyłce. Wyjaśnienia tego stanu rzeczy mogą być różne, ale jak to zwykle bywa najprostsze wyjaśnienia są najbardziej trafne, a jak nie wiadomo o co chodzi to chodzi o pieniądze. Jednym z nich jest wyjaśnienie, iż test *PCMark* korzysta ze specjalnych dla konkretnego procesora optymalizacji, stąd po ich zamianie ustawienia były "źle" dobrane, a wyniki nieoptymalne. Test ten jednak pochodzi z 2005 roku, podczas gdy procesory *Nano* firmy *VIA* pojawiły się w połowie 2008 roku, nie mógł on więc posiadać jakichkolwiek optymalizacji dla tego procesora nawet jakby bardzo się starał. idąc dalej założmy, że owy test posiada te "udogodnienia" dla każdego z procesorów. Dlaczego więc ustawiając "zły" identyfikator zamiast pogorszyć wyniki polepszyliśmy je i nie o kilka lecz o kilkadziesiąt procent?

Oczywiście twórcy *PCMark* zarzekają się jeden przez drugiego, że nie ma tu miejsca żadna manipulacja, że to najpewniej błąd w ich oprogramowaniu, którego zaraz usuną i tak dalej, ale chyba jasno widać na czyją korzyść błąd działa. Poza tym jeżeli owa aplikacja posiada jakiegokolwiek ułatwienia dla któregośkolwiek z procesorów, to dyskwalifikuje to ją jako miarodajne narzędzie do mierzenia wydajności czegokolwiek, gdyż test powinien być taki sam dla wszystkich, bez ułatwień ani utrudnień.

Można tak wymieniać długo, przez kolejne wersje sterowników graficznych coraz to lepiej radzących sobie z testami *3DMark* (z resztą tego samego producenta co *PCMark*) oraz różne inne mniejsze i większe oszustwa w drodze do chwalenia się większą wydajnością. Jak widać łatwo jest wybrać nieodpowiednie narzędzia, których wyniki nie tylko nic nam nie powiedzą, ale na dodatek zniekształcą wyniki i to dość znacznie. Przede wszystkim musimy po prostu pamiętać co chcemy zmierzyć i co mierzy nasz test. Dobrze jeżeli dany benchmark jest dostępny razem z całym kodem źródłowym, jednak specyfika rynku aplikacji na systemach Windows

”preferuje” aplikacje binarne i mało kto pyta tutaj o kod źródłowy. Należy jednak sprawdzić, czy aplikacja której chcemy użyć nie zawiera niechcianych niespodzianek jak *PCMark*.

Na koniec zastanówmy się co jest tak naprawdę ważne jeżeli chodzi o wydajność wirtualizacji. Tak naprawdę można owe zagadnienie podzielić na dwie części. Wydajność wirtualizacji systemów z rodziny Windows, gdyż to na nich jest aktualnie dostępne najwięcej oprogramowania, łącznie z oprogramowaniem kooperacyjnym i specjalistycznym, a często dostępnym tylko na ten system. Do tego worka należy także dorzucić ogólne pojęcie wydajności pełnej wirtualizacji systemów operacyjnych w najlepszym wypadku z asystą parawirtualnych sterowników.

Drugą rządzącą się zupełnie innymi prawami jest wydajność wirtualizacji rozwiązań serwerowych. Tutaj najczęściej jest stosowana parawirtualizacja oraz wirtualizacja na poziomie systemu operacyjnego, gdyż zapewniają one największą wydajność oraz pozwalają na współdzielenie wielu systemów guest na jednej maszynie przez mały ”narzut” wirtualizacji.

Rozdział 4

Przyszłość wirtualizacji

Mówiąc kolokwialnie przyszłość wirtualizacji jest jasna. Rynek nigdy wcześniej nie oferował tak wielu różnorodnych rozwiązań (w dodatku za darmo), nawet do celów korporacyjnych. Projekty *open source* na bieżąco wymieniają się nowymi usprawnieniami i rozwiązaniami zapewniając stały rozwój. Producenci procesorów w praktycznie każdym procesorze implementują swoje rozszerzenia sprzętowe już teraz zapowiadając następne generacje rozszerzeń, zapewniające jeszcze większą wydajność oraz funkcjonalność.

Jednym z przyszłych zastosowań wirtualizacji będzie z pewnością *cluster virtualization* (czyli klastry wirtualizacji). Wiele komputerów czy też wydajnych serwerów połączonych w jedną logiczną jednostkę, z logicznymi procesorami (lub nawet jednym) wykorzystującymi moc wszystkich składowych procesorów do konkretnego zadania. Z czasem powstaną mechanizmy wykorzystujące technologię *GPGPU* do obliczeń maszyny wirtualnej, a z pojawieniem się produktu *Larrabee* stanie się to jeszcze prostsze ze względu na kompatybilność tego procesora graficznego z instrukcjami *i386* i brak konieczności przepisywania wszystkiego na przykład na rozwiązanie *CUDA*.

Innym rozwiązaniem, które prawdopodobnie zyska popularność w przyszłości będzie uruchamianie maszyny wirtualnej dla każdego z programów (w podobny sposób jak rozwiązanie przeglądarki *Google Chrome*, gdzie każda zakładka jest osobnym procesem). Zapewni to niespotykane dotąd bezpieczeństwo. Gdy upewnimy się, że aplikacja nie stanowi żadnego zagrożenia dla naszego systemu, będziemy mogli zdecydować czy chcemy uruchamiać ją "natywnie". Gdyby już teraz wyposażać systemy operacyjne w tego rodzaju *sandbox* wirusy przestałyby być zagrożeniem, gdyż nie byłyby w stanie w jakikolwiek sposób wpłynąć na pracę systemu, wszystkie skutki ich negatywnego działania pozostałyby w maszynie wirtualnej. Rozwiązanie to można by połączyć z implementacją antywirusa przez co przy pierwszym uruchomieniu aplikacji będziemy wiedzieli czy nie zawiera ona jakichś przykrych niespodzianek.

Zastosowań wirtualizacji w przyszłości jest bardzo wiele, z pewnością będzie ona bardziej powszechna i dostępna, oraz stosowana na jeszcze większą skalę niż ma to miejsce dziś.

Rozdział 5

Podsumowanie

Celem mojej pracy było opisanie, zestawienie i porównanie wszystkich liczących się rozwiązań w dziedzinie wirtualizacji systemów operacyjnych. Przedstawienie w przejrzysty sposób wszystkich stosowanych mechanizmów i technologii używanych w dzisiejszych produktach oraz podzielenie ich na odpowiednie kategorie. Praca ta, to swoisty przekrój wachlarza dostępnych dzisiaj rozwiązań w dziedzinie wirtualizacji. Zawiera także opis rozwiązań bardziej unikatowych i mniej popularnych, które będą miały znacznie większe znaczenie w bliższej przyszłości tej dziedziny informatyki.

Nie sposób pominąć tutaj podrozdziału poświęconego rozszerzeniom sprzętowym stworzonym w celu wspierania wirtualizacji. Jest ich coraz więcej oraz są one coraz bardziej wyspecjalizowane, stworzone z myślą dostarczenia jednej konkretnej funkcjonalności w najlepszy możliwy sposób. Oczywiście rozszerzenia te, podobnie jak rozwój samych hypervisorów również ewoluują i w nowszych procesorach implementowane są rozszerzenia następnej generacji, jeszcze lepiej realizujące swoje zadania.

Materiał tutaj przedstawiony może z powodzeniem być bazą do dalszego zgłębiania wiedzy w temacie wirtualizacji, porównywania implementacji konkretnych technologii i zastosowań. Może także służyć jako swego rodzaju paleta dostępnych rozwiązań w dziedzinie wirtualizacji systemów operacyjnych. Należy pamiętać, że wirtualizacja to wyjątkowo dynamicznie rozwijająca się gałąź informatyki, więc z czasem część zawartego tutaj materiału z ulegnie przedawnieniu. Praca uchwyci najistotniejsze aspekty tematyki w aktualnym stadium rozwoju dzięki czemu za parę lat będzie można zweryfikować które drogi rzeczywiście okazały się krokiem na przód, a które porzucono.

Jest to charakterystyczne dla rozwoju techniki i nauki. Dla postępu danej dziedziny wiedzy mają znaczenie każde badania i wszystkie próby jej udoskonalenia. Nie jest najistotniejsze, które z wariantów okażą się w przyszłości najlepsze, ale właśnie sam proces ich odkrywania i doskonalenia oraz wynik prac wszystkich związanych z nim projektantów i programistów.

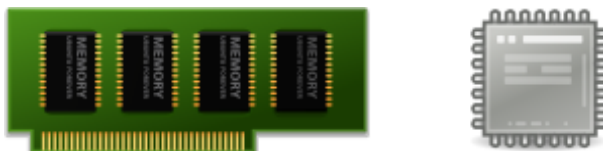
Technikalia

Problemy z wyświetlaniem obrazków

W razie problemów z wyświetlaniem rysunków na systemach *Windows* należy pobrać i zainstalować wtyczkę `svgviewer.exe` firmy *Adobe* stąd <http://www.adobe.com/svg/viewer/install/main.html>, inaczej obrazki które są w formacie **SVG** będą bardzo poszarpane i często nieczytelne.

Użyte ikony projektu Tango

W powyższej pracy zmodyfikowałem i użyłem ikon projektu *Tango*, którego strona znajduje się na <http://tango.freedesktop.org>. Zgodnie z licencją projektu, czyli *Creative Commons* [Creative Commons, 2002] jestem zobligowany do zwrócenia zmian, co też uczyniłem wysyłając owe modyfikacje na listę mailingową projektu tango-artists@lists.freedesktop.org. Owe modyfikacje użyte są w rysunkach 1.30, 1.31, 1.33 oraz 1.32.



Oryginały użytych ikon z projektu *Tango*.

Użyte oprogramowanie

Napewno wiele osób chciałoby wiedzieć w "czym" powstała owa praca, oto więc lista użytego oprogramowania. Sama praca jest zbudowana za pomocą systemu \LaTeX (konkretnie pakiet `tetex` na systemach UNIX) a bibliografia za pomocą biblioteki `BibTeX`. Do wygenerowania indeksu zaś posłużył program `makeindex`. Wektorowe obrazki w formacie **SVG** zostały stworzone w programie *Inkspace*, wszystkie pozostałe obrazki powstały w programie *GIMP*, obydwa te narzędzia dostępne są na otwartej licencji *GPL* i można je pobrać odpowiednio z <http://inkscape.org> oraz <http://gimp.org>.

Bibliografia

- K. Adams / O. Agesen. *Comparison of Software and Hardware Techniques for x86 Virtualization*. Oct 2006.
- AVAXIO. *Virtualizációs technikák*. 2007. URL http://avaxio.hu/virttech_intro.
- J. Stoess / C. Lang / F. Bellosa. *Energy Management for Hypervisor-Based Virtual Machines*. Jun 2007. URL <http://i30www.ira.uka.de/research/publications/pm>.
- Berkeley Computer Systems Research Group of the University of California. *Licencja BSD*. 1989. URL <http://opensource.org/licenses/bsd-license.php>.
- Creative Commons. *Licencja Creative Commons 2.5*. Dec 2002. URL <http://creativecommons.org/licenses/by-sa/2.5>.
- V. Uhlig / J. LeVasseur / E. Skoglund / U. Dannowski. *Towards Scalable Multiprocessor Virtual Machines*. May 2004. URL <http://l4ka.org/publications>.
- J. de Gelas. *Hardware Virtualization: the Nuts and Bolts*. Mar 2008a. URL <http://it.anandtech.com/printarticle.aspx?i=3263>.
- J. de Gelas. *Secrets of Virtual Benchmarking*. Aug 2008b. URL <http://it.anandtech.com/weblog/showpost.aspx?i=479>.
- N. Tolia / M. Satyanarayanan / E. de Lara / H. A. Lagar-Cavilla. *VMM Independent Graphics Acceleration*. 2006. URL <http://cs.toronto.edu/~andreslc/vmgl>.
- Ubuntu Documentation. *Seamless Virtualization*. 2008. URL <https://help.ubuntu.com/community/SeamlessVirtualization>.
- J. M. Holzer / D. Dutile / B. Donahue. *RHEL Para-virtualized Drivers Guide*. 2008.
- S. Hand / A. Warfield / K. Fraser. *Hardware Virtualization with Xen*. Login Press, Feb 2007.
- G. Gerzon. *Intel® Processor Virtualization Extensions*. 2007. URL <http://intel.com>.
- G. J. Popek / R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, Jul 1974.
- O. Goldshmidt. *Virtualization Advanced Operating Systems*. 2007.

- B. Leslie / G. Heiser. *Towards Untrusted Device Drivers*. Mar 2003.
- V. Uhlig / J. LeVasseur / G. Heiser. Are virtual-machine monitors microkernels done right? *ACM SIGOPS Operating Systems Review*, 40(1):95–99, Jan 2006.
- J. G. Hansen / A. K. Henriksen. Nomadic operating systems. Master’s thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 2002. URL <http://nomadbios.dk>.
- J. Hruska. *Low-end Grudge Match: Nano vs. Atom*. Jul 2008. URL <http://arstechnica.com/reviews/hardware/atom-nano-review.ars/6>.
- IBM. *IBM Systems Virtualization*. Dec 2005.
- J. S. Robin / C. E. Irvine. *Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor*. Aug 2000. URL <http://cistr.nps.navy.mil>.
- I. Kelly. *Porting MINIX to Xen*. May 2006.
- H. A. Lagar-Cavilla. *VMGL: VMM-Independent Graphics Acceleration*. May 2007. URL <http://cs.toronto.edu/~andreslc>.
- D. Magenheimer. *Memory Overcommit... Without the Commitment*. Jun 2008.
- J. Nakajima / A. K. Mallick. *Hybrid Virtualization Enhanced Virtualization for Linux*. Jun 2007.
- Sun Microsystems. *VirtualBox Architecture*. 2008. URL http://virtualbox.org/wiki/VirtualBox_architecture.
- T. Ormandy. *An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments*. 2007.
- AMD White Paper. *Live Migration with AMD-V Extended Migration Technology*. 2007. URL <http://amd.com>.
- J. Stoess / C. Lang / M. Reinhardt. *Energy-aware Processor Management for Virtual Machines*. Apr 2006. URL <http://i30www.ira.uka.de/research/publications/pm/>.
- A. Przywara / J. Rödel. *Linux Mainframe - KVM and the Road of Virtualization*. May 2007.
- M.A. Salsburg. *What’s All the Fuss About I/O Virtualization?* Jun 2007. URL http://cmg.org/measureit/issues/mit42/m_42_2.html.
- SPEC. *SPECjbb*. Standard Performance Evaluation Corporation, 2005. URL <http://spec.org/jbb2005>.
- R. M. Stallman. *Licencja GPL*. 1989. URL <http://gnu.org/licenses/gpl.html>.
- Sun. *Licencja CDDL*. 2004. URL <http://sun.com/cddl/cddl.html>.

- D. McIlroy / J. Ossanna / D. Ritchie / K. Thompson. *UNIX time-sharing operating system*. 1969. URL <http://wikipedia.org/wiki/UNIX>.
- V. Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Germany, May 2005.
- Jeff Victor. *How to Move a Solaris Container*. 2008. URL http://sun.com/software/solaris/howtoguides/moving_containers.jsp.
- VMware. *Virtualization: Architectural Considerations and Other Evaluation Criteria*. 2005.
- VMware. *Performance Benchmarking Guidelines for VMware Workstation 5.5*. 2006.
- VMware. *Understanding Full Virtualization, Paravirtualization and Hardware Assist*. 2007.
- E. Wahlig. *Hardware Based Virtualization Technologies*. May 2008.
- M. Ben-Yehuda / J. Mason / O. Krieger / J. Xenidis / L. Van Doorn / A. Mallick / J. Nakajima / E. Wahlig. *Using IOMMUs for Virtualization in Linux and Xen*. Jul 2006.
- L. Sanger / J. Wales. *Wikipedia*. 2001. URL <http://wikipedia.org>.

Spis rysunków

1.1	Schemat działania systemu operacyjnego.	6
1.2	Typowe wykorzystanie <i>protection rings</i> przez system operacyjny.	7
1.3	<i>Emulator</i> jest po prostu kolejną aplikacją działającą w systemie.	8
1.4	Schemat <i>hypervisora typu 1</i> na procesorze bez obsługi <i>monitor mode</i>	11
1.5	Schemat <i>hypervisora typu 1</i> na procesorze z obsługą <i>monitor mode</i>	12
1.6	Schemat <i>hypervisora typu 2</i> na mechanizmie <i>protection rings</i>	12
1.7	<i>Parawirtualizacja</i> oraz <i>pełna wirtualizacja</i> przy użyciu <i>hypervisora typu 1</i>	13
1.8	<i>Pełna wirtualizacja</i> realizowana za pomocą <i>hypervisora typu 2</i>	16
1.9	Schemat <i>wirtualizacji na poziomie systemu</i>	16
1.10	Schemat działania rozwiązania <i>hardware partitions</i>	18
1.11	Schemat działania <i>pamięci wirtualnej</i>	19
1.12	Schemat działania układu <i>memory management unit</i>	19
1.13	Schemat translacji przy użyciu <i>page table</i>	20
1.14	Schemat translacji przy użyciu bufora <i>TLB</i>	20
1.15	Schemat mechanizmu <i>shadow page table</i>	21
1.16	Schemat mechanizmu <i>nested page table</i>	22
1.17	Schemat dostępu do urządzenia <i>natywnie</i>	24
1.18	Porównanie działania układów <i>MMU</i> oraz <i>I/O MMU</i>	24
1.19	Programowe emulowanie układu <i>IOMMU</i> przez <i>hypervisor</i>	25
1.20	Schemat sprzętowego układu <i>IOMMU</i>	26
1.21	Schemat dzielenia urządzeń za pomocą <i>programowych urządzeń wirtualnych</i>	27
1.22	Schemat dzielenia urządzeń za pomocą <i>rozwiązania sprzętowego</i>	27
1.23	Schemat <i>programowego "przekazywania"</i> urządzeń do systemu <i>guest</i>	28
1.24	Schemat dzielenia urządzeń za pomocą <i>rozwiązania sprzętowego</i>	28
1.25	Logo procesorów firmy <i>Advanced Micro Devices</i>	29
1.26	Logo <i>HyperTransport Consortium</i>	31
1.27	Logo grupy <i>Trusted Computing Group</i>	33
1.28	Logo procesorów firmy <i>Intel</i>	34
1.29	Logo procesorów firmy <i>VIA</i>	37
1.30	Wykorzystanie szyny <i>FSB</i> w konfiguracji <i>SMP</i>	40
1.31	Technologia <i>NUMA</i> w konfiguracji <i>SMP</i>	41
1.32	Sposób komunikacji rdzeni w procesorach <i>Core 2 Quad</i>	43

1.33	Sposób komunikacji rdzeni w procesorach architektury <i>K10</i>	43
1.34	Jednostki wykonawcze procesora z włączoną/wyłączoną obsługą <i>HTT</i>	44
1.35	Wirtualizacja systemu guest w oknie.	49
1.36	Wirtualizacja w trybie <i>seamless mode</i>	49
2.1	Wynik polecenia <code>xm list</code>	57
2.2	Konsola zarządzająca monitorem <i>VMware ESX</i>	58
2.3	Schemat rozwiązania <i>Parallels Virtuozzo Containers</i>	70
2.4	Rozwiązanie <i>CoLinux</i> działające pod kontrolą systemu <i>Windows</i>	72

Indeks

Address Space Identifier, 20, 30
algorytm szeregowania, 41
ALU, 43
AMD-V, 12, 14, 28, 55, 60, 63, 64
amd64, 60
ASID, 30

bare metal, 54
binary translation, 11–14, 57–59, 61, 62, 64
BIOS, 17
Bochs, 7, 72
branded zones, 67
BrandZ, 67
BSD, 4
buffer overflow, 65
buffer underflow, 66
bug, 17

cache, 12, 14, 18, 40
Cache Coherent NUMA, 40
ccNUMA, 40
CDDL, 4
chroot, 16
clipboard, 62
cluster virtualization, 76
coherence, 47, 62
cold boot, 73
coLinux, 70
containers, 16
content based page sharing, 22
Cool'n'Quiet, 30, 31
CoolCore, 31
Cooperative Linux, 70
Cooperative Virtual Machine, 70
copy on write, 22, 58, 68

CR3, 19
Crossbow, 59
CUDA, 46

DCA, 29
DEV, 28
Device Exclusion Vector, 28
Direct Connect Architecture, 29
direct execution, 12, 14, 58, 59, 61, 62, 64
direct recompilation, 62
DMA, 23
DMA remapping, 23
dom0, 54
domU, 54
drag and drop, 47, 62
DTrace, 67
Dual Dynamic Power Management, 31
dynamic recompilation, 12, 58, 59, 64

EIST, 34
emulacja, 7
Enhanced Power Management, 31
ESX, 56
Extended Migration, 29
Extended Page Tables, 34
ezjail, 66

firmware, 17
FlexMigration, 35
FlexPriority, 34
FPU, 43
FreeBSD, 41
FSB, 28–30, 34, 39

Gallium3D, 47
GART, 24

- gates, 6
- global zone, 67
- GPGPU, 25
- GPL, 4, 60
- GSX Server, 61
- guest, 4
- guest CR3, 20
- hardware, 4
- hardware page table pointer, 19
- hardware partitions, 16
- host, 4
- host CR3, 20
- hosted hypervisor, 11, 59
- hot boot, 73
- HTT, 42
- HVM, 55
- hybrydowa wirtualizacja, 16
- Hyper Threading Technology, 42
- hypercall, 13
- HyperTransport, 29, 30
- Hypervisor, 9
- Hypervisor typu 1, 9
- Hypervisor typu 2, 11
- I/O, 6, 8
- I/O Acceleration Technology, 33
- I/O Memory Management Unit, 29
- I/O MMU, 23
- I/OAT, 34
- idle, 34
- IMC, 28
- Independent Dynamic Core, 31
- input, 31, 35
- Instruction Set Architecture, 8
- Integrated Memory Controller, 28
- Intel VT-x, 14
- Inter Process Communication, 40
- IOMMU, 23, 29, 47
- IPC, 40
- ISA, 8
- Isaiah, 36

- iSCSI, 59
- ISO, 46
- JAVA, 38
- just in time, 12, 14
- K10, 36, 42
- Kernel based Virtual Machine, 60
- kernel mode, 6, 8
- konsolidacja, 17
- kontenery, 16
- kqemu, 57, 58
- KVM, 12, 60
- LaGrande, 35
- Larrabee, 46
- Linux binary compatibility, 66
- Linux OpenVZ, 68
- Linux VServer, 67
- live migration, 35
- logical domain, 16
- Mac OS X, 7
- mapowanie DMA, 23
- Maszyna wirtualna, 8
- memory ballooning, 22
- memory management unit, 18
- memory overcommitment, 21
- microkernel, 6
- MINIX 3, 6
- MMU, 18
- monitor maszyny wirtualnej, 8
- monitor mode, 10
- natywna wirtualizacja, 14
- NCQ, 59
- ncurses, 69
- NDA, 44
- nested page table, 20, 29, 34
- Non Disclosure Agreement, 44
- non root mode, 10, 28, 33
- Non-Uniform Memory Access, 39
- Nouveau, 45

NUMA, 29, 39, 40
 Open Computing Language, 46
 open source, 58
 OpenAL, 46
 OpenCL, 46
 OpenGL, 45, 46
 Optimized Power Management, 30
 OS level virtualization, 15
 OS/2, 6
 Pacifica, 28
 page table, 18
 pages, 18
 paging, 18
 pamięć, 4
 pamięć wirtualna, 18
 para, 12
 Parallels Virtuozzo Containers, 68, 69
 parawirtualizacja, 12, 16
 partycje sprzętowe, 16
 patch, 67
 pełna wirtualizacja, 10, 14, 16
 PearPC, 7
 physics processing, 46
 PIO, 23
 point-to-point, 34
 PowerNow, 30, 31
 Presidio, 31
 procesor, 4
 properitary, 45
 protection rings, 5, 9
 QEMU, 7, 55, 57, 59, 60, 72
 QEMU Lanucher, 58
 Qemulator, 58
 Qmuranet, 60
 QNX, 6
 QuickPath Interconnect, 34
 qvm86, 57
 Rapid Virtualization Indexing, 29
 ray tracing, 46
 RDP, 48, 59
 real time operating system, 18
 Remote Desktop Protocol, 48
 reverse engineering, 45
 ring -1, 10
 ring 0, 5
 ring 1, 5
 ring 2, 5
 ring 3, 5
 root mode, 10, 28, 33
 RTOS, 18
 sandbox, 65, 68
 SBE, 56
 scan before execution, 56
 scheduler, 41
 seamless mode, 47, 59, 62
 server entity, 16
 shadow page table, 20
 Single-Root Input/Output Virtualization, 34
 SmartSelect, 63
 SMP, 28
 snapshot/commit, 62
 Solaris Containers, 59, 66
 Solaris Zones, 66
 sparse zones, 67
 SPECjbb, 73
 SR-IOV, 34
 sterowniki parawirtualne, 38
 Stream, 46
 stronicowanie, 18
 SVM, 28
 swap, 22, 60
 swapping, 22
 syscall, 13, 67, 68
 system operacyjny, 4
 Systems Network Architecture, 30
 tablica stron, 18
 Tagged TLB, 20, 30, 35
 task priority register, 34
 TLB, 18

topology aware scheduler, 41
 translation lookaside buffer, 18
 transparent page sharing, 22
 Trusted Computing Group, 31, 35
 Trusted Execution Technology, 35
 Trusted Platform Module, 31, 35
 Tungsten Graphics, 47
 TV-x, 63, 64
 TXT, 35

 ULE, 41
 UML, 70
 unified shaders, 46
 USB, 59, 62
 user mode, 6, 8
 User Mode Linux, 70
 userspace, 5, 10, 16

 Vanderpool, 32
 VIA, 36
 VIA NANO, 36
 virtual environment, 16
 Virtual Machine Device Queues, 33
 Virtual Machine Interface, 13
 Virtual Machine Manager, 60
 virtual machine monitor, 8, 9
 virtual memory, 18
 Virtual Processor Identifier, 35
 Virtual Router Redundancy Protocol, 50
 VirtualBox, 58
 Virtualization Technology for Connectivity, 33
 Virtualization Technology for Directed I/O, 33
 VMDq, 33
 VMGL, 45
 VMI, 13, 57, 61
 VMM, 8
 VMware, 56
 VMware Server, 61
 VMware Workstation, 61
 VMX root, 10
 VPID, 35
 VT-c, 33
 VT-d, 33, 47
 VT-i, 33
 VT-x, 12, 33, 55, 60

 Win4BSD, 65
 Win4Solaris, 65
 WINE, 63
 WireGL, 45
 wirtualizacja, 4, 8
 wirtualizacja na poziomie systemu, 15
 wirtualizacja operacji I/O, 22
 wirtualizacja pamięci, 17
 wirtualizacja wspierana sprzętowo, 14
 wirtualne środowisko, 16

 Xen, 54, 55
 xVM, 55

 ZFS, 67
 zones, 67