Virtualização: Conceitos e Aplicações em Segurança

Article

4 authors, including:

Carlos Maziero
Universidade Federal do Paraná
77 PUBLICATIONS **233** CITATIONS

SEE PROFILE

Capítulo 4

Virtualização: Conceitos e Aplicações em Segurança

Marcos Aurelio Pchek Laureano[1,2], Carlos Alberto Maziero[1]

[1]Programa de Pós-Graduação em Informática
Pontifícia Universidade Católica do Paraná
Curitiba PR

[2]Centro Universitário Franciscano (UNIFAE)
Curitiba PR

Abstract

The recent evolution of the virtual machines' technology allowed their large-scale adoption in production systems. Currently, the main use of virtual machines has been the servers' consolidation, aiming at reducing hardware, software, and management costs of the installed computing infrastructure. However, in the recent years, several research works showed many possibilities of using virtual machines in the systems security area. This work presents the fundamental concepts about virtual machines, discusses advantages and drawbacks of using virtual machines in several application domains, explores the usage possibilities of virtual machines in the systems security area, and shows some examples of popular virtual machine environments.

Resumo

A evolução recente da tecnologia de máquinas virtuais permitiu a sua adoção em larga escala nos sistemas de produção. O principal uso de máquinas virtuais tem sido a consolidação de servidores e, conseqüentemente, a redução de custos em hardware, software e gerência do parque tecnológico. No entanto, várias pesquisas vêm demonstrando possibilidades de aplicação de máquinas virtuais na área de segurança de sistemas. Este trabalho apresenta os fundamentos conceituais sobre máquinas virtuais, discute as vantagens e desvantagens no uso de máquinas virtuais, explora as principais possibilidades de uso de máquinas virtuais na área de segurança de sistemas e expõe alguns exemplos de ambientes de máquinas virtuais de uso corrente.

4.1. Introdução à virtualização

O conceito de máquina virtual não é recente. Os primeiros passos na construção de ambientes de máquinas virtuais começaram na década de 1960, quando a IBM desenvolveu o sistema operacional experimental M44/44X. A partir dele, a IBM desenvolveu vários sistemas comerciais suportando virtualização, entre os quais o famoso OS/370 [Goldberg 1973, Goldberg and Mager 1979]. A tendência dominante nos sistemas naquela época era fornecer a cada usuário um ambiente mono-usuário completo, com seu próprio sistema operacional e aplicações, completamente independente e desvinculado dos ambientes dos demais usuários.

Na década de 1980, com a popularização de plataformas de hardware baratas como o PC, a virtualização perdeu importância. Afinal, era mais barato, simples e versátil fornecer um computador completo a cada usuário, que investir em sistemas de grande porte, caros e complexos. Além disso, o hardware do PC tinha desempenho modesto e não provia suporte adequado à virtualização, o que inibiu o uso de ambientes virtuais nessas plataformas.

Com o aumento de desempenho e funcionalidades do hardware PC e o surgimento da linguagem Java, no início dos anos 90, o interesse pelas tecnologias de virtualização voltou à tona. Apesar da plataforma PC Intel ainda não oferecer um suporte adequado à virtualização, soluções engenhosas como as adotadas pela empresa VMWare permitiram a virtualização nessa plataforma, embora com desempenho relativamente modesto. Atualmente, as soluções de virtualização de linguagens e de plataformas vêm despertando grande interesse do mercado. Várias linguagens são compiladas para máquinas virtuais portáveis e os processadores mais recentes trazem um suporte nativo à virtualização.

4.1.1. Arquitetura dos sistemas computacionais

Uma máquina real é formada por vários componentes físicos que fornecem operações para o sistema operacional e suas aplicações. Iniciando pelo núcleo do sistema real, o processador central (CPU) e o *chipset* da placa-mãe fornecem um conjunto de instruções e outros elementos fundamentais para o processamento de dados, alocação de memória e processamento de entrada/saída. Os sistemas de computadores são projetados com basicamente três componentes: hardware, sistema operacional e aplicações. O papel do hardware é executar as operações solicitadas pelas aplicações através do sistema operacional. O sistema operacional recebe as solicitações das operações (por meio das chamadas de sistema) e controla o acesso ao hardware – principalmente nos casos em que os componentes são compartilhados, como o sistema de memória e os dispositivos de entrada/saída.

Os sistemas de computação convencionais são caracterizados por níveis de abstração crescentes e interfaces bem definidas entre eles. Um dos principais objetivos dos sistemas operacionais é oferecer uma visão abstrata, de alto nível, dos recursos de hardware, que sejam mais simples de usar e menos dependente das tecnologias subjacentes. As abstrações oferecidas pelo sistema às aplicações são construídas de forma incremental, em níveis separados por interfaces bem definidas e relativamente padronizadas. Cada interface encapsula as abstrações dos níveis inferiores, permitindo assim o desenvolvimento independente dos vários níveis, o que simplifica a construção e evolução dos sistemas.

Exemplos típicos dessa estruturação em níveis de abstração crescentes, separados por interfaces bem definidas, são os subsistemas de rede e de disco em um sistema operacional convencional. No sub-sistema de arquivos, cada nível de abstração trata de um problema: interação com o dispositivo físico de armazenamento, escalonamento de acessos ao dispositivo, gerência de *buffers* e *caches*, alocação de arquivos, diretórios, controle de acesso, etc. A figura 4.1 apresenta os níveis de abstração de um subsistema de gerência de disco típico.

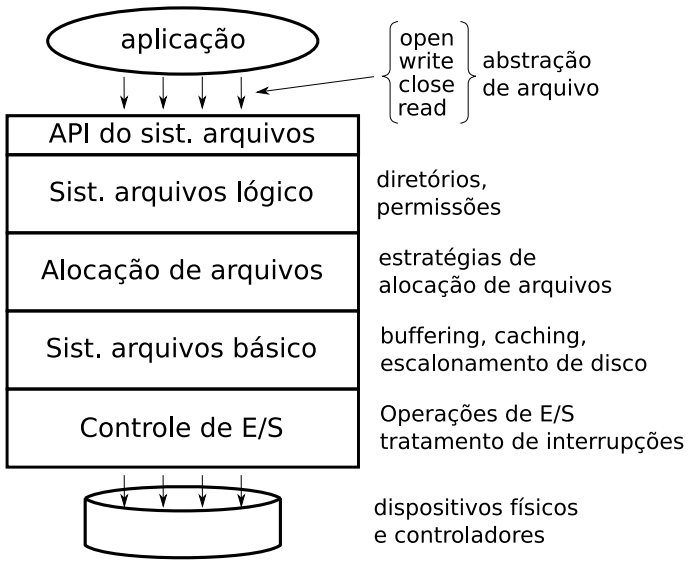

Figura 4.1. Níveis de abstração em um sub-sistema de disco.

As interfaces existentes entre os componentes de um sistema de computação são:

- *Conjunto de instruções* (*ISA – Instruction Set Architecture*): é a interface básica entre o hardware e o software, sendo constituída pelas instruções em código de máquina aceitas pelo processador e todas as operações de acesso aos recursos do hardware (acesso físico à memória, às portas de entrada/saída, ao relógio do sistema, etc.). Essa interface é dividida em duas partes:
  - *Instruções de usuário* (*User ISA*): compreende as instruções do processador e demais itens de hardware acessíveis aos programas do usuário, que executam com o processador operando em modo não-privilegiado;
  - *Instruções de sistema* (*System ISA*): compreende as instruções do processador e demais itens de hardware, unicamente acessíveis ao núcleo do sistema operacional, que executa em modo privilegiado;
- *Chamadas de sistema* (*syscalls*): é o conjunto de operações oferecidas pelo núcleo do sistema operacional aos processos dos usuários. Essas chamadas permitem um acesso controlado das aplicações aos dispositivos periféricos, à memória e às instruções privilegiadas do processador.

- *Chamadas de bibliotecas* (*libcalls*): bibliotecas oferecem um grande número de funções para simplificar a construção de programas; além disso, muitas chamadas de biblioteca encapsulam chamadas do sistema operacional, para tornar seu uso mais simples. Cada biblioteca possui uma interface própria, denominada *Interface de Programação de Aplicações* (*API – Application Programming Interface*). Exemplos típicos de bibliotecas são a *LibC* do UNIX (que oferece funções como `fopen` e `printf`), a GTK+ (*Gimp ToolKit*, que permite a construção de interfaces gráficas) e a SDL (*Simple DirectMedia Layer*, para a manipulação de áudio e vídeo).

A figura 4.2 apresenta essa visão conceitual da arquitetura de um sistema computacional, com seus vários componentes e as respectivas interfaces entre eles.

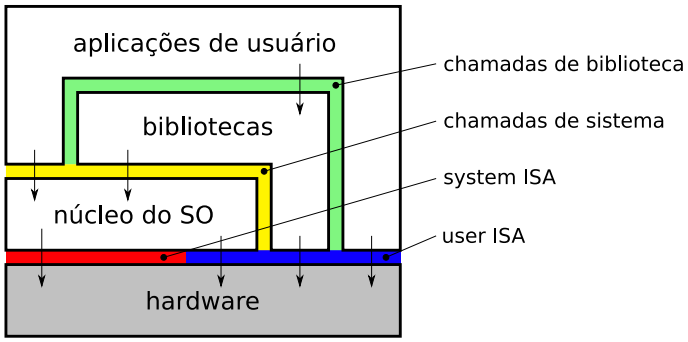

Figura 4.2. Componentes e interfaces de um sistema computacional.

4.1.2. Compatibilidade entre interfaces de sistema

Para que programas e bibliotecas possam executar sobre uma determinada plataforma, é necessário que tenham sido compilados para ela, respeitando o conjunto de instruções do processador em modo usuário (*User ISA*) e o conjunto de chamadas de sistema oferecido pelo sistema operacional. A visão conjunta dessas duas interfaces (*User ISA* + *syscalls*) é denominada *Interface Binária de Aplicação* (*ABI – Application Binary Interface*). Da mesma forma, um sistema operacional só poderá executar sobre uma plataforma de hardware se tiver sido construído e compilado de forma a respeitar sua interface ISA (*User/System ISA*). A figura 4.3 representa essas duas interfaces.

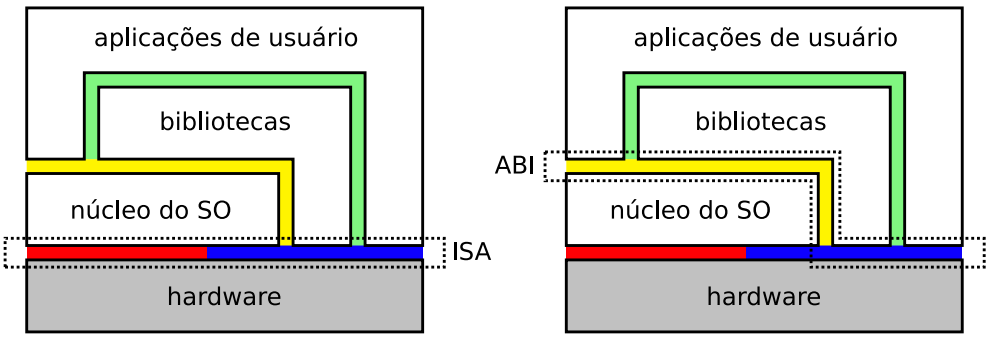

Figura 4.3. Interfaces de sistema ISA e ABI [Smith and Nair 2004].

Nos sistemas computacionais de mercado atuais, as interfaces de baixo nível ISA e ABI são normalmente fixas, ou pouco flexíveis. Geralmente não é possível criar novas instruções de processador ou novas chamadas de sistema operacional, ou mesmo mudar sua semântica para atender às necessidades específicas de uma determinada aplicação. Mesmo se isso fosse possível, teria de ser feito com cautela, para não comprometer o funcionamento de outras aplicações.

Os sistemas operacionais, assim como as aplicações, são projetados para aproveitar o máximo dos recursos que o hardware fornece. Normalmente os projetistas de hardware, sistema operacional e aplicações trabalham de forma independente (em empresas e tempos diferentes). Por isso, esses trabalhos independentes geraram, ao longo dos anos, várias plataformas computacionais diferentes e incompatíveis entre si.

Observa-se então que, embora a definição de interfaces seja útil, por facilitar o desenvolvimento independente dos vários componentes do sistema, torna pouco flexíveis as interações entre eles: um sistema operacional só funciona sobre o hardware (ISA) para o qual foi construído, uma biblioteca só funciona sobre a ABI para a qual foi projetada e uma aplicação tem de obedecer a ABIs/APIs pré-definidas. A figura 4.4, extraída de [Smith and Nair 2004], ilustra esses problemas de compatibilidade entre interfaces.

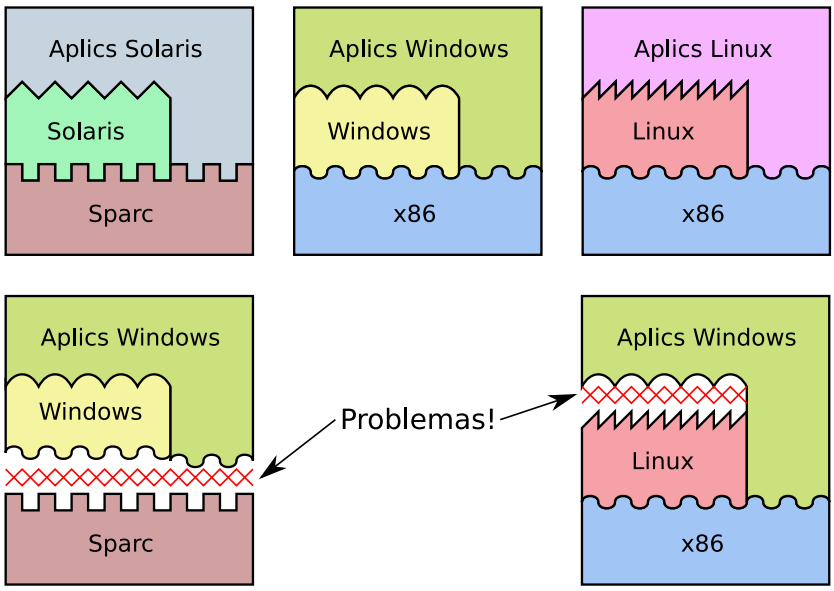

Figura 4.4. Problemas de compatibilidade entre interfaces [Smith and Nair 2004].

A baixa flexibilidade na interação entre as interfaces dos componentes de um sistema computacional traz vários problemas [Smith and Nair 2004]:

- *Baixa portabilidade*: a mobilidade de código e sua interoperabilidade são requisitos importantes dos sistemas atuais, que apresentam grande conectividade de rede e diversidade de plataformas. A rigidez das interfaces de sistema atuais dificulta sua construção, por acoplar excessivamente as aplicações aos sistemas operacionais e aos componentes do hardware.

- *Barreiras de inovação*: a presença de interfaces rígidas dificulta a construção de novas formas de interação entre as aplicações e os dispositivos de hardware (e com os usuários, por conseqüência). Além disso, as interfaces apresentam uma grande inércia à evolução, por conta da necessidade de suporte às aplicações já existentes.

- *Otimizações inter-componentes*: aplicações, bibliotecas, sistemas operacionais e hardware são desenvolvidos por grupos distintos, geralmente com pouca interação entre eles. A presença de interfaces rígidas a respeitar entre os componentes leva cada grupo a trabalhar de forma isolada, o que diminui a possibilidade de otimizações que envolvam mais de um componente.

Essas dificuldades levaram à investigação de outras formas de relacionamento entre os componentes de um sistema computacional. Uma das abordagens mais promissoras nesse sentido é o uso de *máquinas virtuais*, que serão apresentadas na próxima seção.

4.1.3. Máquinas virtuais

Conforme visto, as interfaces padronizadas entre os componentes do sistema de computação permitem o desenvolvimento independente dos mesmos, mas também são fonte de problemas de interoperabilidade, devido à sua pouca flexibilidade. Por isso, não é possível executar diretamente em um processador Intel/AMD uma aplicação compilada para um processador ARM: as instruções em linguagem de máquina do programa não serão compreendidas pelo processador Intel. Da mesma forma, não é possível executar diretamente em Linux uma aplicação escrita para Windows, pois as chamadas de sistema emitidas pelo programa não serão compreendidas pelo sistema operacional subjacente.

Todavia, é possível contornar esses problemas de compatibilidade através de uma *camada de virtualização* construída em software. Usando os serviços oferecidos por uma determinada interface de sistema, é possível construir uma camada de software que ofereça aos demais componentes uma outra interface. Essa camada de software permitirá o acoplamento entre interfaces distintas, de forma que um programa desenvolvido para a plataforma *A* possa executar sobre uma plataforma distinta *B* (figura 4.5).

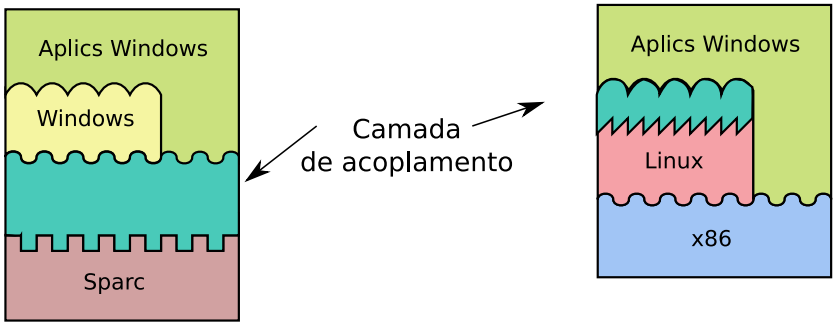

Figura 4.5. Acoplamento entre interfaces distintas

Usando os serviços oferecidos por uma determinada interface de sistema, a camada de virtualização constrói outra interface de mesmo nível, de acordo com as necessidades dos componentes de sistema que farão uso dela. A nova interface de sistema,

vista através dessa camada de virtualização, é denominada *máquina virtual*. A camada de virtualização em si é denominada *hipervisor* ou *monitor de máquina virtual*.

A figura 4.6, extraída de [Smith and Nair 2004], apresenta um exemplo de máquina virtual, onde um hipervisor permite executar um sistema operacional Windows e suas aplicações sobre uma plataforma de hardware Sparc, distinta daquela para a qual esse sistema operacional foi projetado (Intel/AMD).

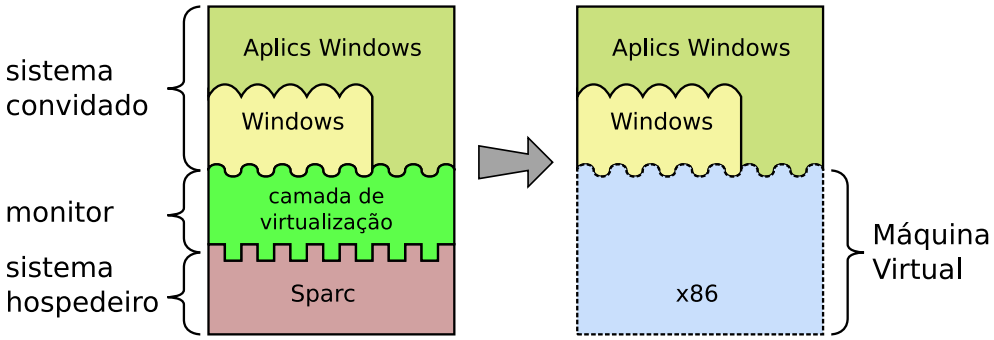

Figura 4.6. Uma máquina virtual [Smith and Nair 2004].

Um ambiente de máquina virtual consiste de três partes básicas, que podem ser observadas na figura 4.6:

- O sistema real, nativo ou hospedeiro (*host system*), que contém os recursos reais de hardware e software do sistema;
- o sistema virtual, também denominado *sistema convidado* (*guest system*), que executa sobre o sistema virtualizado; em alguns casos, vários sistemas virtuais podem coexistir, executando simultaneamente sobre o mesmo sistema real;
- a camada de virtualização, *hipervisor*, ou *monitor* (VMM – *Virtual Machine Monitor*), que constrói as interfaces virtuais a partir da interface real.

Originalmente, uma máquina virtual era definida como uma cópia eficiente, isolada e protegida de uma máquina real [Popek and Goldberg 1974, Goldberg and Mager 1979]. Essa definição, vigente nos anos 1960-70, foi construída a partir da perspectiva de um sistema operacional: uma abstração de software que gerencia um sistema físico (máquina real). Com o passar dos anos, o termo "máquina virtual" evoluiu e englobou um grande número de abstrações, como por exemplo a *Java Virtual Machine* (JVM), que não virtualiza um sistema real [Rosenblum 2004].

A construção de máquinas virtuais é bem mais complexa que possa parecer à primeira vista. Caso os conjuntos de instruções do sistema real e do sistema virtual sejam diferentes, é necessário usar as instruções da máquina real para simular as instruções da máquina virtual. Além disso, é necessário mapear os recursos de hardware virtuais (periféricos oferecidos ao sistema convidado) sobre os recursos existentes na máquina real (os periféricos reais). Por último, pode ser necessário mapear as chamadas de sistema

emitidas pelas aplicações do sistema convidado em chamadas equivalentes no sistema real, quando os sistemas operacionais virtual e real forem distintos.

Existem várias formas de implementar a virtualização, que serão descritas nas próximas seções. Algumas delas são apresentadas sucintamente na figura 4.7, como:

- *Virtualização completa*: um sistema operacional convidado e suas aplicações, desenvolvidas para uma plataforma de hardware *A*, são executadas sobre uma plataforma de hardware distinta *B*.
- *Emulação do sistema operacional*: as aplicações de um sistema operacional *X* são executadas sobre outro sistema operacional *Y*, na mesma plataforma de hardware.
- *Tradução dinâmica*: as instruções de máquina das aplicações são traduzidas durante a execução em outras instruções mais eficientes para a mesma plataforma.
- *Replicação de hardware*: são criadas várias instâncias virtuais de um mesmo hardware real, cada uma executando seu próprio sistema operacional convidado e suas respectivas aplicações.

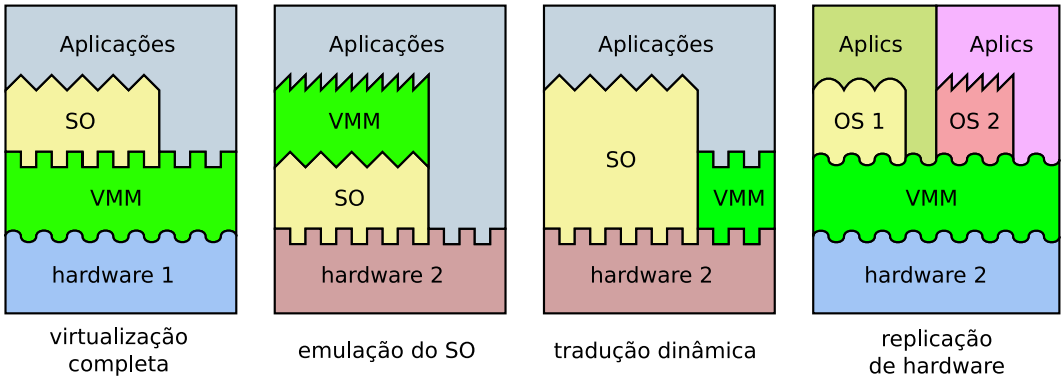

Figura 4.7. Possibilidades de virtualização [Smith and Nair 2004].

4.1.3.1. Abstração versus virtualização

Embora a virtualização possa ser vista como um tipo de abstração, existe uma clara diferença entre os termos "abstração" e "virtualização", no contexto de sistemas [Smith and Nair 2005]. A abstração de recursos consiste em fornecer uma interface de acesso homogênea e simplificada ao recursos do sistema. Por outro lado, a virtualização consiste em criar novas interfaces a partir das interfaces existentes. Na virtualização, os detalhes de baixo nível da plataforma real não são necessariamente ocultos, como ocorre na abstração de recursos. A figura 4.8 ilustra essa diferença: através da virtualização, um processador Sparc pode ser visto como um processador Intel. Da mesma forma, um disco real no padrão SATA pode ser visto como vários discos menores independentes, com a mesma interface (SATA) ou outra interface (IDE).

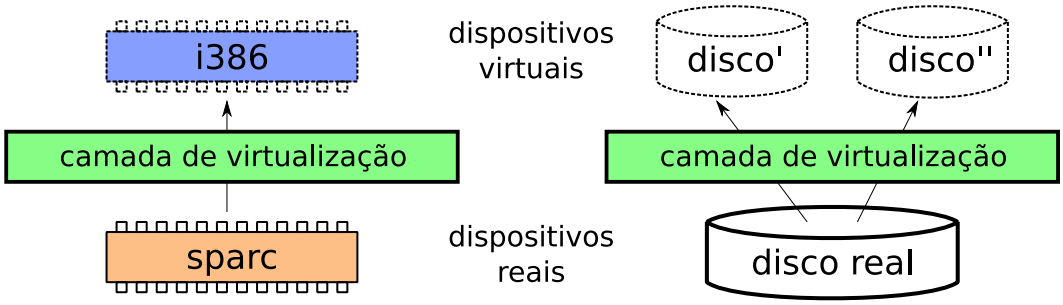

Figura 4.8. Virtualização versus abstração de recursos.

A figura 4.9 ilustra outro exemplo dessa diferença no contexto do armazenamento em disco. A abstração provê às aplicações o conceito de "arquivo", sobre o qual estas podem executar operações simples como `read` ou `write`, por exemplo. Já a virtualização fornece para a camada superior apenas um disco virtual, construído a partir de um arquivo do sistema operacional real subjacente. Esse disco virtual terá de ser particionado e formatado para seu uso, da mesma forma que um disco real.

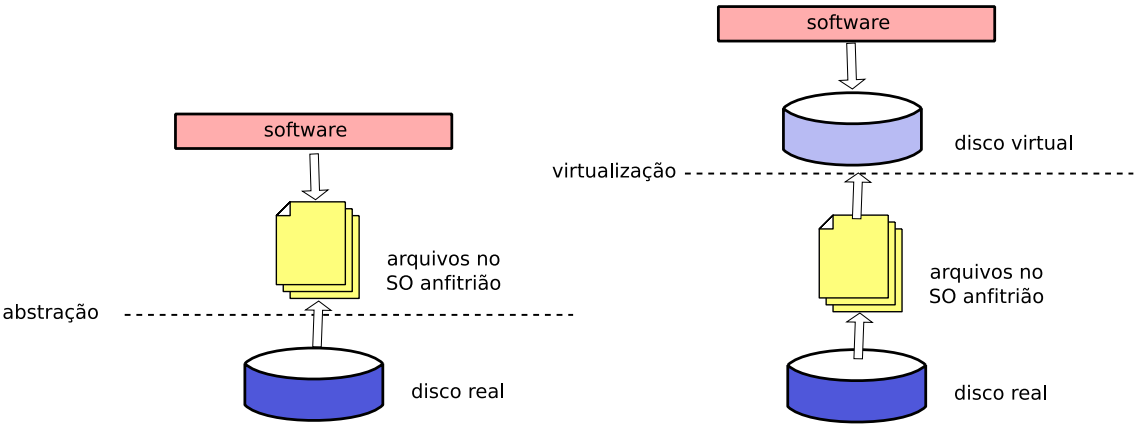

Figura 4.9. Abstração versus virtualização de um disco rígido.

4.1.3.2. Propriedades dos hipervisores

Para funcionar de forma correta e eficiente, um hipervisor deve atender a alguns requisitos básicos: ele deve prover um ambiente de execução aos programas essencialmente idêntico ao da máquina real, do ponto de vista lógico. Programas executando sobre uma máquina virtual devem apresentar, no pior caso, leves degradações de desempenho. Além disso, o hipervisor deve ter controle completo sobre os recursos do sistema real (o sistema hospedeiro). A partir desses requisitos, foram estabelecidas por [Goldberg 1973, Popek and Goldberg 1974] as seguintes propriedades a serem satisfeitas por um hipervisor ideal:

- *Equivalência*: um hipervisor provê um ambiente de execução quase idêntico ao da máquina real original. Todo programa executando em uma máquina virtual deve se

comportar da mesma forma que o faria em uma máquina real; exceções podem resultar somente de diferenças nos recursos disponíveis (memória, disco, etc), dependências de temporização e a existência dos dispositivos de entrada/saída necessários à aplicação.

- *Controle de recursos*: o hipervisor deve possuir o controle completo dos recursos da máquina real: nenhum programa executando na máquina virtual deve possuir acesso a recursos que não tenham sido explicitamente alocados a ele pelo hipervisor, que deve intermediar todos os acessos. Além disso, a qualquer instante o hipervisor pode resgatar recursos previamente alocados.

- *Eficiência*: grande parte das instruções do processador virtual (o processador provido pelo hipervisor) deve ser executada diretamente pelo processador da máquina real, sem intervenção do hipervisor. As instruções da máquina virtual que não puderem ser executadas pelo processador real devem ser interpretadas pelo hipervisor e traduzidas em ações equivalentes no processador real. Instruções simples, que não afetem outras máquinas virtuais ou aplicações, podem ser executadas diretamente no hardware.

Além dessas três propriedades básicas, as propriedades derivadas a seguir são freqüentemente associadas a hipervisores [Popek and Goldberg 1974, Rosenblum 2004]:

- *Isolamento*: esta propriedade garante que um software em execução em uma máquina virtual não possa ver, influenciar ou modificar outro software em execução no hipervisor ou em outra máquina virtual. Esta propriedade pode ser utilizada para garantir que erros de software ou aplicações maliciosas possam ser contidos em um ambiente controlado (uma máquina virtual), sem afetar outras partes do sistema.

- *Inspeção*: o hipervisor tem acesso e controle sobre todas as informações do estado interno da máquina virtual, como registradores do processador, conteúdo de memória, eventos etc.

- *Gerenciabilidade*: como cada máquina virtual é uma entidade independente das demais, a administração de diversas instâncias de máquinas virtuais sobre um mesmo supervisor é simplificada e centralizada. O hipervisor deve ter mecanismos para gerenciar o uso dos recursos existentes entre os sistemas convidados.

- *Encapsulamento*: como o hipervisor tem acesso e controle sobre o estado interno de cada máquina virtual em execução, ele pode salvar *checkpoints* de uma máquina virtual, periodicamente ou em situações especiais (por exemplo, antes de uma atualização de sistema operacional). Esses *checkpoints* são úteis para retornar a máquina virtual a estados anteriores (*rollback*), para análises *post-mortem* em caso de falhas, ou para permitir a migração da máquina virtual entre hipervisores executando em computadores distintos.

- *Recursividade*: alguns sistemas de máquinas virtuais exibem também esta propriedade: deve ser possível executar um hipervisor dentro de uma máquina virtual, produzindo um novo nível de máquinas virtuais. Neste caso, a máquina real é normalmente denominada *máquina de nível 0* (figura 4.10).

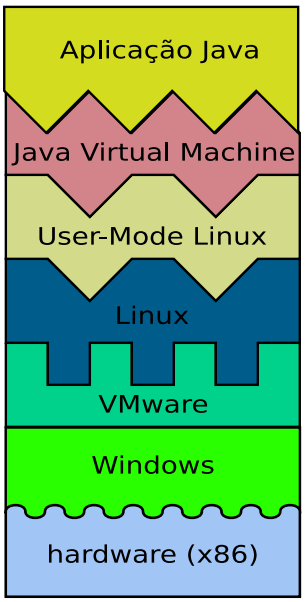

Figura 4.10. Diversos níveis de virtualização [Laureano 2006]

Essas propriedades também podem ser utilizadas na segurança de sistemas [Honeynet Project 2003, Garfinkel and Rosenblum 2003] e outras aplicações. As propriedades básicas caracterizam um hipervisor ideal, que nem sempre pode ser construído sobre as plataformas de hardware existentes. A possibilidade de construção de um hipervisor em uma determinada plataforma é definida através do seguinte teorema, enunciado e provado por Popek e Goldberg em [Popek and Goldberg 1974]:

> *Para qualquer computador convencional de terceira geração, um hipervisor pode ser construído se o conjunto de instruções sensíveis daquele computador for um sub-conjunto de seu conjunto de instruções privilegiadas.*

Para compreender melhor as implicações desse teorema, é necessário definir claramente os seguintes conceitos:

- *Computador convencional de terceira geração*: qualquer sistema de computação convencional seguindo a arquitetura de Von Neumann, que suporte memória virtual e dois modos de operação: modo usuário e modo privilegiado.
- *Instruções sensíveis*: são aquelas que podem consultar ou alterar o status do processador, ou seja, os registradores que armazenam o status atual da execução na máquina real;
- *Instruções privilegiadas*: são acessíveis somente por meio de códigos executando em nível privilegiado (código de núcleo). Caso um código não-privilegiado tente executar uma instrução privilegiada, uma exceção (interrupção) é gerada, ativando uma rotina de tratamento previamente especificada pelo núcleo do sistema real.

De acordo com esse teorema, toda instrução sensível deve ser também privilegiada. Assim, quando uma instrução sensível for executada por um programa não-privilegiado (um núcleo convidado ou uma aplicação convidada), provocará a ocorrência de uma interrupção. Essa interrupção pode ser usada para ativar uma *rotina de interpretação* dentro do hipervisor, que irá simular o efeito da instrução sensível (ou seja, interpretá-la), de acordo com o contexto onde sua execução foi solicitada (máquina virtual ou hipervisor). Obviamente, quanto maior o número de instruções sensíveis, maior o volume de interpretação de código a realizar, e menor o desempenho da máquina virtual.

No caso de processadores que não atendam o teorema de Popek e Goldberg, podem existir instruções sensíveis que executem sem gerar interrupções, o que impede o hipervisor de interceptá-las e interpretá-las. Uma solução possível para esse problema é a *tradução dinâmica* das instruções sensíveis acessíveis nos programas de usuário: ao carregar um programa na memória, o hipervisor analisa seu código e substitui as instruções problemáticas por chamadas a rotinas que as interpretam dentro do hipervisor. Isso implica em um tempo maior para o lançamento de programas, mas torna possível a virtualização. Outra técnica possível para resolver o problema é a *para-virtualização*. Estas técnicas são discutidas na seção 4.1.5.

4.1.3.3. Suporte de hardware

Na época em que Popek e Goldberg definiram seu principal teorema, o hardware virtualizável dos mainframes IBM suportava parcialmente as condições impostas pelo mesmo. Esses sistemas dispunham de uma funcionalidade chamada *execução direta*, que permitia a uma máquina virtual acessar nativamente o hardware para execução de instruções. Esse mecanismo permitia que aqueles sistemas obtivessem, com a utilização de máquinas virtuais, desempenho similar ao de sistemas convencionais equivalentes [Goldberg 1973, Popek and Goldberg 1974, Goldberg and Mager 1979].

O suporte de hardware necessário para a construção de hipervisores eficientes está presente em sistemas de grande porte, mas é apenas parcial nos micro-processadores de mercado. Por exemplo, a família de processadores *Intel Pentium IV* (e anteriores) possui 17 instruções sensíveis que podem ser executadas em modo usuário sem gerar exceções, o que viola o teorema de Goldberg e dificulta a criação de máquinas virtuais em sistemas que usam esses processadores [Robin and Irvine 2000].

Por volta de 2005, os principais fabricantes de micro-processadores (*Intel* e *AMD*) incorporaram um suporte básico à virtualização em seus processadores, através das tecnologias IVT (*Intel Virtualization Technology*) e AMD-V (*AMD Virtualization*), que são conceitualmente equivalentes [Uhlig et al. 2005]. A idéia central de ambas as tecnologias consiste em definir dois modos possíveis de operação do processador: os modos *root* e *non-root*. O modo *root* equivale ao funcionamento de um processador convencional, e se destina à execução de um hipervisor. Por outro lado, o modo *non-root* se destina à execução de máquinas virtuais. Além de Intel e AMD, outros fabricantes de hardware têm se preocupado com o suporte à virtualização. Em 2005, a *Sun Microsystems* incorporou suporte nativo à virtualização em seus processadores *UltraSPARC* [Yen 2007]. Em 2007, a IBM propôs uma especificação de interface de hardware denominada *IBM Power ISA*

2.04 [IBM 2007], que respeita os requisitos necessários à virtualização do processador e da gestão de memória.

4.1.4. Tipos de máquinas virtuais

Conforme discutido na seção 4.1.3, existem diversas possibilidades de implementação de sistemas de máquinas virtuais, cada uma com suas características próprias de eficiência e equivalência. De acordo com o tipo de sistema convidado suportado, os ambientes de máquinas virtuais podem ser divididos em duas grandes famílias (figura 4.11):

- *Máquinas virtuais de aplicação* (*Process Virtual Machines*): são ambientes de máquinas virtuais destinados a suportar apenas um processo ou aplicação convidada específica. A máquina virtual Java e o ambiente de depuração *Valgrind* são exemplos desse tipo de ambiente.
- *Máquinas virtuais de sistema* (*System Virtual Machines*): são ambientes de máquinas virtuais construídos para suportar sistemas operacionais convidados completos, com aplicações convidadas executando sobre eles. Como exemplos, temos os ambientes *VMware* e *VirtualBox*.

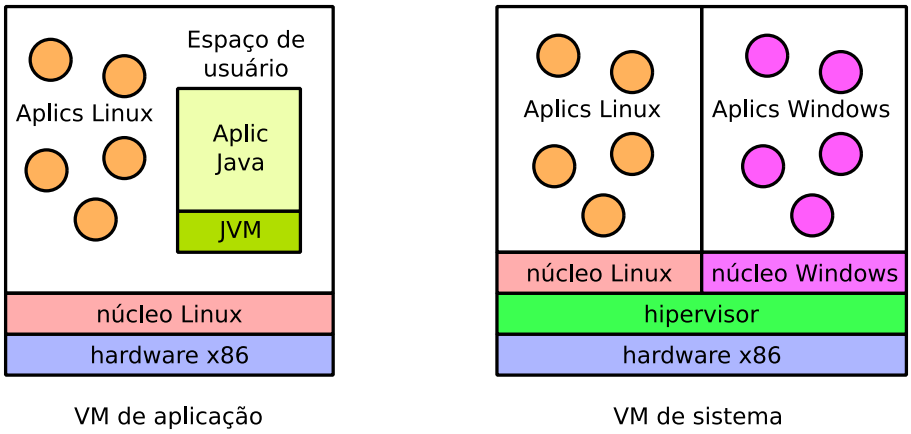

Figura 4.11. Máquinas virtuais de aplicação e de sistema.

Por outro lado, os ambientes de máquinas virtuais também podem ser classificados de acordo com o nível de similaridade entre as interfaces de hardware do sistema convidado e do sistema real (ISA):

- *Interfaces equivalentes*: a interface virtual oferecida ao ambiente convidado reproduz a interface de hardware do sistema real, permitindo a execução de aplicações construídas para o sistema real. Como a maioria das instruções do sistema convidado pode ser executada diretamente pelo processador (com exceção das instruções sensíveis), o desempenho obtido pelas aplicações convidadas pode ser próximo do desempenho de execução no sistema real. Ambientes como *VMware* são exemplos deste tipo de ambiente.

- *Interfaces distintas*: a interface virtual não tem relação com a interface de hardware do sistema real, ou seja, implementa um conjunto de instruções distinto, a ser interpretado pelo hipervisor. A interpretação de instruções impõe um custo de execução significativo ao sistema convidado. O emulador *QEMU*, que provê às aplicações um processador virtual *Intel Pentium II*, é um exemplo desta abordagem.

A virtualização também pode ser classificada como [Rosenblum 2004]:

- *Virtualização do hardware*: a virtualização exporta o sistema físico como hardware abstrato (semelhante ao sistema original). Neste modelo, qualquer software escrito para a arquitetura nativa (*x*86, por exemplo) irá funcionar no sistema convidado. Este foi o modelo adotado na década de 60 para o VM/370 nos mainframes IBM e é a tecnologia de virtualização utilizado pelo *VMware* na plataforma *x*86.
- *Virtualização do sistema operacional*: a virtualização exporta um sistema operacional como abstração de um sistema específico. A máquina virtual executa aplicações – ou um conjunto de aplicações – de um sistema operacional específico. O *FreeBSD Jail* ou o *User-Mode Linux* são exemplos desta tecnologia.
- *Virtualização de linguagens de programação*: a camada de virtualização cria uma aplicação no topo do sistema operacional. Na prática, as máquinas virtuais nesta categoria são desenvolvidas para computadores fictícios projetados para uma finalidade específica. A camada exporta uma abstração para a execução de programas escritos para esta virtualização. *Java JVM* e *Smalltalk* são exemplos deste tipo de máquina virtual.

O trabalho [Nanda and Chiueh 2005] ainda classifica a virtualização como:

- *Abstração da ISA* (*Instruction Set Architecture*): a virtualização é implementada com o uso da emulação completa da ISA. O emulador executa as instruções do sistema convidado (a máquina virtual é obtida por meio da emulação) utilizando a tradução das instruções para o sistema nativo. Esta arquitetura é robusta e simples para implementação, mas a perda de desempenho é significativa. *Bochs*, *Crusoe* e *QEMU* são exemplos da mesma.
- *Hardware Abstraction Layer* (HAL): o hipervisor simula uma arquitetura completa para o sistema convidado. Desta forma, o sistema convidado acredita estar executando sobre um sistema completo de hardware. *VMware*, *Virtual PC*, *Denali* e *Xen* são exemplos desta arquitetura.
- *OS Level* (sistema operacional): este nível de virtualização é obtido utilizando uma chamada de sistema (*system call*) específica. O principal benefício da virtualização neste nível é criar uma camada para obter o isolamento de processos, cada sistema é virtualizado com seu próprio endereço IP e outros recursos de hardware (embora limitados). A virtualização ocorre a partir de um diretório ou sistema de arquivos separadado e previamente preparado para este fim. O *FreeBSD Jail* é um exemplo desta arquitetura.

- *Nível de aplicação* ou virtualização de linguagens de programação: a virtualização é obtida por meio da abstração de uma *camada de execução*. Uma aplicação utiliza esta camada para executar as instruções do programa. Esta solução garante que uma aplicação possa ser executada em qualquer plataforma de software ou hardware, pois a camada é abstraída de forma idêntica em todas as plataformas. Todavia, torna-se necessária uma máquina virtual específica para cada plataforma. *Java JVM*, *Microsoft .NET CLI* e *Parrot* são exemplos desta arquitetura.

- *User level library interface* (biblioteca de interface para usuário): vários sistemas e aplicações são escritos utilizando um conjunto de APIs fornecidos pelo sistema (aplicações sob o sistema Windows são os exemplos mais populares), exportados para o nível do usuário por meio de bibliotecas. A *virtualização* neste nível é obtida com a abstração do topo do sistema operacional, para que as aplicações possam executar em outra plataforma. O *Wine* é um exemplo deste tipo de arquitetura.

4.1.4.1. Máquinas Virtuais de Aplicação

Uma máquina virtual de aplicação (ou *Process Virtual Machine*) suporta a execução de um processo ou aplicação individual. Ela é criada sob demanda, no momento do lançamento da aplicação convidada, e destruída quando a aplicação finaliza sua execução. O conjunto *hipervisor + aplicação* é visto então como um único processo dentro do sistema operacional subjacente, submetido às mesmas condições e restrições que os demais processos nativos.

Os hipervisores que implementam máquinas virtuais de aplicação normalmente permitem a interação entre a aplicação convidada e as demais aplicações do sistema, através dos mecanismos usuais de comunicação e coordenação entre processos, como mensagens, *pipes* e semáforos. Além disso, também permitem o acesso normal ao sistema de arquivos e outros recursos locais do sistema. Ao criar a máquina virtual para uma aplicação, o hipervisor pode implementar a mesma interface de hardware (ISA) da máquina real subjacente, ou implementar uma interface distinta. Quando a interface da máquina real é preservada, boa parte das instruções do processo convidado podem ser executadas diretamente, com exceção das instruções sensíveis, que devem ser interpretadas pelo hipervisor. Os exemplos mais comuns de máquinas virtuais de aplicação que preservam a interface ISA real são os *sistemas operacionais multi-tarefas*, os *tradutores dinâmicos* e alguns *depuradores de memória*:

- *Sistemas operacionais multi-tarefas*: os sistemas operacionais que suportam vários processos simultâneos também podem ser vistos como ambientes de máquinas virtuais. Em um sistema multi-tarefas, cada processo recebe um *processador virtual* (simulado através de fatias de tempo do processador real), uma *memória virtual* (através do espaço de endereços mapeado para aquele processo) e *recursos físicos* (acessíveis através de chamadas de sistema). Este ambiente de máquinas virtuais é tão antigo e tão presente em nosso cotidiano que costumamos ignorá-lo. No entanto, ele simplifica muito a tarefa dos programadores, que não precisam se preocupar com a gestão do compartilhamento desses recursos entre os processos.

- *Tradutores dinâmicos*: um tradutor dinâmico consiste em um hipervisor que analisa e otimiza um código executável, para tornar sua execução mais rápida e eficiente. A otimização não muda o conjunto de instruções da máquina real usado pelo código, apenas reorganiza as instruções de forma a acelerar sua execução. Por ser dinâmica, a otimização do código é feita durante a carga do processo na memória ou durante a execução de suas instruções, de forma transparente. O artigo [Duesterwald 2005] apresenta uma descrição detalhada desse tipo de abordagem.

- *Depuradores de memória*: alguns sistemas de depuração de erros de acesso à memória, como o sistema *Valgrind* [Seward and Nethercote 2005], executam o processo sob depuração em uma máquina virtual. Todas as instruções do programa que manipulam acessos à memória são executadas de forma controlada, a fim de encontrar possíveis erros. Ao depurar um programa, o sistema *Valgrind* inicialmente traduz seu código em um conjunto de instruções interno, manipula esse código para inserir operações de verificação de acessos à memória e traduz o código modificado de volta ao conjunto de instruções da máquina real, para em seguida executá-lo e verificar os acessos à memória realizados.

As máquinas virtuais de aplicação mais populares hoje são aquelas em que a interface binária de aplicação (ABI) requerida pela aplicação é diferente da oferecida pela máquina real. Como a ABI é composta pelas chamadas do sistema operacional e as instruções de máquina disponíveis à aplicação (*user ISA*), as diferenças podem ocorrer em ambos. Nos dois casos, o hipervisor terá de realizar traduções dinâmicas (durante a execução) das ações requeridas pela aplicação em suas equivalentes na máquina real (como visto, um hipervisor com essa função é denominado *tradutor dinâmico*).

Caso as diferenças de interface entre aplicação e máquina real se restrinjam às chamadas do sistema operacional, o hipervisor precisa apenas mapear as chamadas de sistema usadas pela aplicação sobre as chamadas oferecidas pelo sistema operacional da máquina real. Essa é a abordagem usada, por exemplo, pelo ambiente *Wine*, que permite executar aplicações Windows em plataformas Unix. As chamadas de sistema Windows emitidas pela aplicação em execução são interceptadas e transformadas em chamadas Unix, de forma dinâmica e transparente (figura 4.12).

Entretanto, muitas vezes a interface ISA utilizada pela aplicação não corresponde a nenhum hardware existente, mas a um hardware simplificado que representa uma máquina abstrata. Um exemplo típico dessa situação ocorre na linguagem Java: um programa escrito em Java, ao ser compilado, gera um código binário específico para uma máquina abstrata denominada *máquina virtual Java* (*JVM – Java Virtual Machine*). A linguagem de máquina executada pela máquina virtual Java é denominada *bytecode* Java, e não corresponde a instruções de nenhum processador real. A máquina virtual deve então interpretar todas as operações do *bytecode*, utilizando as instruções da máquina real subjacente para executá-las.

A vantagem mais significativa da abordagem adotada por Java é a *portabilidade* do código executável: para que uma aplicação Java possa executar sobre uma determinada plataforma, basta que a máquina virtual Java esteja disponível ali (na forma de um suporte de execução denominado JRE – *Java Runtime Environment*). Assim, a portabilidade dos

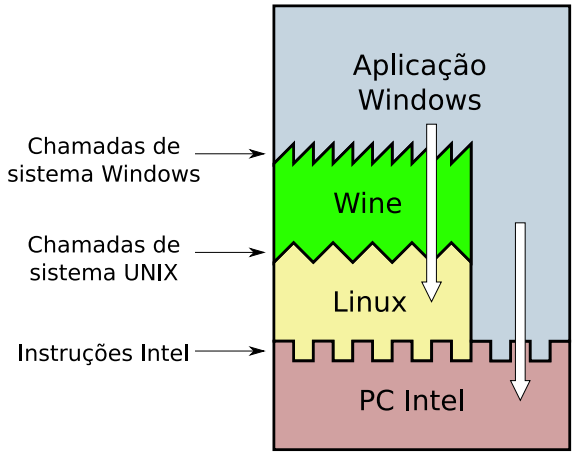

Figura 4.12. Funcionamento do emulador Wine.

programas Java depende unicamente da portabilidade da própria máquina virtual Java. O suporte de execução Java pode estar associado a um navegador Web, o que permite que o código Java seja associado a páginas Web, na forma de pequenas aplicações denominadas *applets*, que são trazidas junto com os demais componentes de página Web e executam localmente no navegador[1].

Em termos de desempenho, um programa compilado para uma máquina virtual executa mais lentamente que seu equivalente compilado sobre uma máquina real, devido ao custo de interpretação do *bytecode*. Todavia, essa abordagem oferece melhor desempenho que linguagens puramente interpretadas. Além disso, técnicas de otimização como a compilação *Just-in-Time* (JIT), na qual blocos de instruções repetidos freqüentemente são compilados pelo hipervisor e armazenados em cache, permitem obter ganhos de desempenho significativos em aplicações executando sobre máquinas virtuais.

4.1.4.2. Máquinas Virtuais de Sistema

Uma máquina virtual de sistema suporta um ou mais sistemas operacionais convidados, com suas respectivas aplicações, que executam de forma isolada e independente. Cada sistema operacional convidado tem a ilusão de executar sozinho sobre uma plataforma de hardware própria.

Em um ambiente virtual, os sistemas operacionais convidados são fortemente isolados uns dos outros, e normalmente só podem interagir através dos mecanismos de rede, como se estivessem em máquinas fisicamente separadas. Todavia, alguns sistemas de máquinas virtuais permitem o compartilhamento controlado de certos recursos. Por exemplo, os sistemas *VMware Workstation* e *VirtualBox* permitem a definição de diretórios compartilhados no sistema de arquivos real, que podem ser acessados pelas máquinas virtuais.

[1]Neste caso, a propriedade de isolamento das máquinas virtuais é aplicada aos *applets* Java, para evitar que códigos maliciosos trazidos da Internet executem ações prejudiciais no sistema local; este mecanismo é conhecido como *Java Sandbox*.

O hipervisor de sistema fornece aos sistemas operacionais convidados uma interface de sistema ISA virtual, que pode ser idêntica ao hardware real, ou distinta. Além disso, ele virtualiza o acesso aos recursos, para que cada sistema operacional convidado tenha um conjunto de recursos virtuais próprio, construído a partir dos recursos físicos existentes na máquina real. Assim, cada máquina virtual terá sua própria interface de rede, seu próprio disco, sua própria memória RAM, etc.

As máquinas virtuais de sistema constituem a primeira abordagem usada para a construção de hipervisores, desenvolvida na década de 1960 e formalizada por Popek e Goldberg [Popek and Goldberg 1974]. Naquela época, a tendência de desenvolvimento de sistemas computacionais buscava fornecer a cada usuário uma máquina virtual com seus recursos virtuais próprios, sobre a qual o usuário executava um sistema operacional mono-tarefa e suas aplicações. Assim, o compartilhamento de recursos não era responsabilidade do sistema operacional convidado, mas do hipervisor subjacente. No entanto, ao longo dos anos 70, como o desenvolvimento de sistemas operacionais multi-tarefas eficientes e seguros como MULTICS e UNIX, as máquinas virtuais de sistema perderam gradativamente seu interesse. Somente no final dos anos 90, com o aumento do poder de processamento dos micro-processadores e o surgimento de novas possibilidades de aplicação, as máquinas virtuais de sistema foram "redescobertas".

Existem vários tipos de ambientes de máquinas virtuais de sistema, que podem ser classificados quanto à sua arquitetura e quanto ao grau de virtualização do hardware. No que diz respeito à arquitetura, existem basicamente dois tipos de hipervisores de sistema, apresentados na figura 4.13:

- *Hipervisores nativos* (ou *de tipo I*): nesta categoria, o hipervisor executa diretamente sobre o hardware da máquina real, sem um sistema operacional subjacente. A função do hipervisor é multiplexar os recursos de hardware (memória, discos, interfaces de rede, etc) de forma que cada máquina virtual veja um conjunto de recursos próprio e independente. Assim, cada máquina virtual se comporta como uma máquina física completa que pode executar o seu próprio sistema operacional. Esta é a forma mais antiga de virtualização, encontrada nos sistemas computacionais de grande porte dos anos 1960-70. Alguns exemplos de sistemas que empregam esta abordagem são o *IBM OS/370*, o *VMware ESX Server* e o ambiente *Xen*.

- *Hipervisores convidados* (ou *de tipo II*): nesta categoria, o hipervisor executa como um processo normal sobre um sistema operacional nativo subjacente. O hipervisor utiliza os recursos oferecidos pelo sistema operacional nativo para oferecer recursos virtuais ao sistema operacional convidado que executa sobre ele. Normalmente, um hipervisor convidado suporta apenas uma máquina virtual com uma instância de sistema operacional convidado. Caso mais máquinas sejam necessárias, mais hipervisores devem ser lançados. Exemplos de sistemas que adotam esta estrutura incluem o *VMware Workstation*, o *QEMU* e o *VirtualBox*.

Pode-se afirmar que os hipervisores convidados são mais flexíveis que os hipervisores nativos, pois podem ser facilmente instalados/removidos em máquinas com sistemas

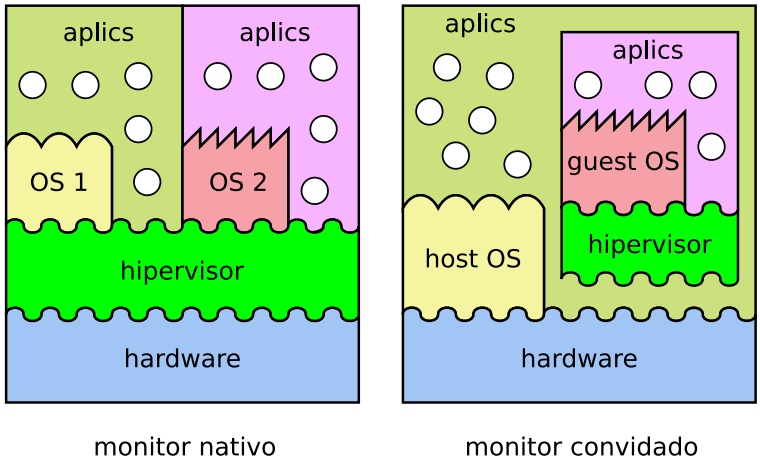

Figura 4.13. Arquiteturas de máquinas virtuais de sistema.

operacionais previamente instalados. Por outro lado, um hipervisor nativo tem melhor desempenho que um hipervisor convidado, pois este último tem de usar os recursos oferecidos pelo sistema operacional subjacente, enquanto o primeiro pode acessar diretamente o hardware real.

No que diz respeito ao nível de virtualização oferecido pelo hipervisor, os ambientes de máquinas virtuais podem ser classificados em duas categorias, conforme apresentado na figura 4.14:

- *Virtualização de recursos*: nesta categoria, a interface ISA de usuário é mantida, apenas as instruções privilegiadas e os recursos (discos, etc) são virtualizados. Dessa forma, o sistema operacional convidado e as aplicações convidadas *vêem* o processador real. Como as quantidades de instruções a virtualizar são pequenas, o desempenho do sistema convidado é próximo daquele obtido, se ele estivesse executando diretamente sobre o hardware real. Os sistemas *VMware Workstation*, *VirtualBox* e *MS VirtualPC* implementam esta estratégia.

- *Virtualização completa*: nesta categoria, toda a interface do hardware é virtualizada, incluindo todas as instruções do processador e os dispositivos de hardware. Isso permite oferecer ao sistema operacional convidado uma interface de hardware distinta daquela fornecida pela máquina real subjacente. O custo de virtualização neste caso é bem maior que na virtualização parcial (de recursos), mas esta abordagem permite executar sistemas operacionais em plataformas distintas daquela para a qual foram inicialmente projetados. Exemplos típicos desta abordagem são os sistemas de máquinas virtuais *QEMU*, que oferece um processador *Intel Pentium II* ao sistema convidado, o *MS VirtualPC for MAC*, que permite executar o sistema Windows sobre uma plataforma de hardware *PowerPC*, e o sistema *Hercules*, que emula um computador *IBM System/390* sobre um PC convencional de plataforma Intel.

Uma categoria especial de hipervisor nativo com virtualização completa consiste nos **hipervisores embutidos no hardware** (*codesigned hypervisors*). Um hipervisor em-

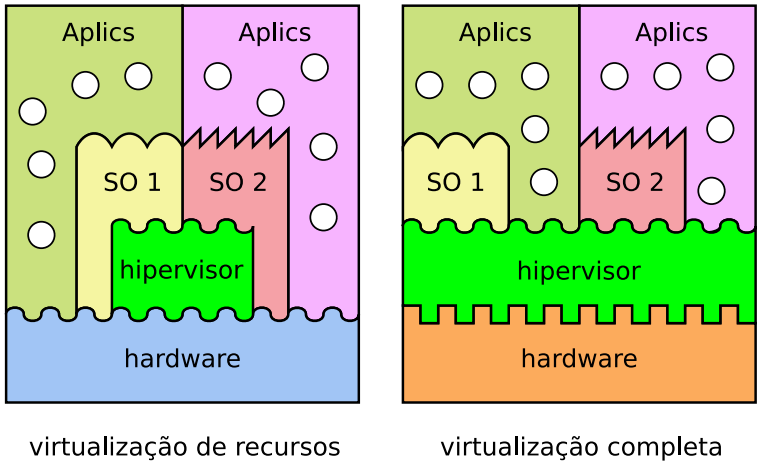

Figura 4.14. Níveis de virtualização: virtualização de recursos e virtualização completa.

butido é visto como parte integrante do hardware da máquina real, e implementa a interface de sistema (ISA) vista pelos sistemas operacionais e aplicações daquela plataforma. O conjunto de instruções do processador real somente está acessível ao hipervisor, que reside em uma área de memória separada da memória principal e usa técnicas de tradução dinâmica para executar as instruções dos sistemas convidados. Um exemplo típico desse tipo de sistema é o processador *Transmeta Crusoe/Efficeon*, que aceita instruções no padrão *Intel 32 bits* e internamente as converte em um conjunto de instruções VLIW (*Very Large Instruction Word*). Como o hipervisor desse processador pode ser reprogramado para criar novas instruções ou modificar as instruções existentes, ele acabou sendo denominado *Code Morphing Software* (figura 4.15).

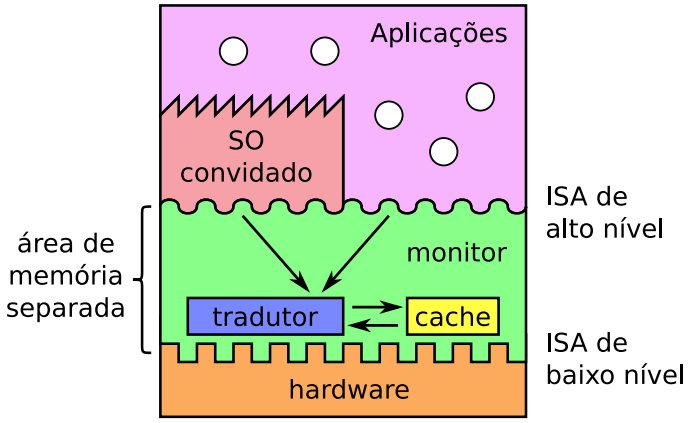

Figura 4.15. Hipervisor embutido no hardware.

A figura 4.16 traz uma classificação dos tipos de máquinas virtuais de sistema, de acordo com os critérios de arquitetura do hipervisor e grau de virtualização oferecido aos sistemas convidados.

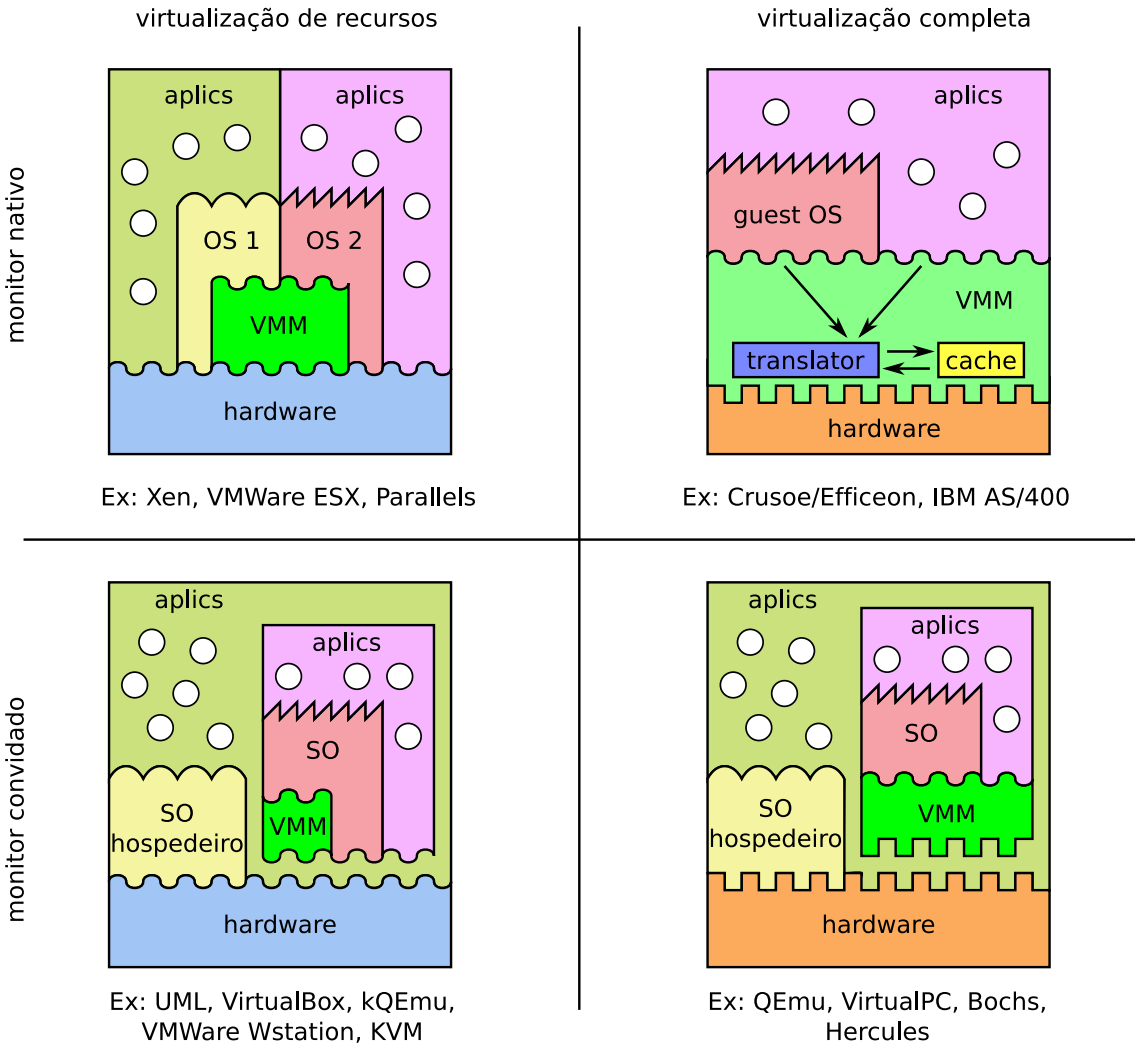

Figura 4.16. Classificação de máquinas virtuais de sistema.

4.1.5. Estratégias de Virtualização

A construção de hipervisores implica na definição de algumas estratégias para a virtualização. As estratégias mais utilizadas atualmente são a virtualização total (*full virtualization*), normalmente associada à tradução dinâmica (*dynamic translation*), e a paravirtualização (*paravirtualization*). Além disso, algumas técnicas complementares são usadas para melhorar o desempenho dos sistemas de máquinas virtuais. Essas técnicas são discutidas nesta seção.

4.1.5.1. Virtualização Total

A virtualização total reconstrói um ambiente virtual no qual o hardware fornecido aos sistemas convidados corresponde a um sistema real existente, Toda a interface de acesso ao hardware é virtualizada, incluindo todas as instruções do processador e os dispositivos de hardware. Desta forma, os sistemas convidados não precisam sofrer nenhum tipo de alteração ou ajuste para executar no ambiente virtual. Esta é a abordagem usada na mai-

oria dos hipervisores de sistema clássicos, como *QEmu* e *VMWare*. Sua maior vantagem consiste em permitir que sistemas operacionais convencionais executem como convidados sem necessidade de modificações. Por outro lado, o sistema convidado executa mais lentamente, uma vez que todos os acessos ao hardware são intermediados pelo hipervisor. Além disso, o hipervisor terá de interceptar e emular todas as instruções sensíveis executadas pelos sistemas convidados, o que tem um custo elevado em plataformas de hardware sem suporte adequado à virtualização.

4.1.5.2. Tradução dinâmica

Uma técnica freqüentemente utilizada na construção de máquinas virtuais é a *tradução dinâmica* (*dynamic translation*) ou recompilação dinâmica (*dynamic recompilation*) de partes do código binário dos sistemas convidados e suas aplicações. Nesta técnica, o hipervisor analisa, reorganiza e traduz as seqüências de instruções emitidas pelo sistema convidado em novas seqüências de instruções, à medida em que a execução do sistema convidado avança.

A tradução binária dinâmica pode ter vários objetivos: (a) adaptar as instruções geradas pelo sistema convidado à interface ISA do sistema real, caso não sejam idênticas; (b) detectar e tratar instruções sensíveis não-privilegiadas (que não geram interrupções ao serem invocadas pelo sistema convidado); ou (c) analisar, reorganizar e otimizar as seqüências de instruções geradas pelo sistema convidado, de forma a melhorar o desempenho de sua execução. Neste último caso, os blocos de instruções muito freqüentes podem ter suas traduções mantidas em cache, para melhorar ainda mais o desempenho.

A tradução dinâmica é usada em vários tipos de hipervisores. Uma aplicação típica é a construção da máquina virtual Java, onde recebe o nome de JIT – *Just-in-Time Bytecode Compiler*. Outro uso corrente é a construção de hipervisores para plataformas sem suporte adequado à virtualização, como os processadores Intel/AMD 32 bits. Neste caso, o código convidado a ser executado é analisado em busca de instruções sensíveis, que são substituídas por chamadas a rotinas apropriadas dentro do supervisor.

No contexto de virtualização, a recompilação dinâmica é composta basicamente dos seguintes passos [Ung and Cifuentes 2006]:

1. *Desmontagem* (*disassembling*): o fluxo de bytes do código convidado a executar é decomposto em blocos de instruções. Cada bloco é normalmente composto de uma seqüência de instruções de tamanho variável, terminando com uma instrução de controle de fluxo de execução;

2. *Geração de código intermediário*: cada bloco de instruções tem sua semântica descrita através de uma representação independente de máquina;

3. *Otimização*: a descrição em alto nível do bloco de instruções é analisada para aplicar eventuais otimizações; como este processo é realizado durante a execução, normalmente somente otimizações com baixo custo computacional são aplicáveis;

4. *Codificação*: o bloco de instruções otimizado é traduzido para instruções da máquina física, que podem ser diferentes das instruções do código original;

5. *Caching*: blocos de instruções com execução muito freqüente têm sua tradução armazenada em cache, para evitar ter de traduzi-los e otimizá-los novamente;

6. *Execução*: o bloco de instruções traduzido é finalmente executado nativamente pelo processador da máquina real.

Esse processo pode ser simplificado caso as instruções de máquina do código convidado sejam as mesmas da máquina real subjacente, o que torna desnecessário traduzir os blocos de instruções em uma representação independente de máquina.

4.1.5.3. Paravirtualização

Em meados dos anos 2000, alguns pesquisadores investigaram a possibilidade de modificar a interface entre o hipervisor e os sistemas convidados, oferecendo a estes um hardware virtual que é similar, mas não idêntico ao hardware real. Essa abordagem, denominada *paravirtualização*, permite um melhor acoplamento entre os sistemas convidados e o hipervisor, o que leva a um desempenho significativamente melhor das máquinas virtuais. As modificações na interface de sistema do hardware virtual (*system ISA*) exigem uma adaptação dos sistemas operacionais convidados, para que estes possam executar sobre a plataforma virtual. Todavia, a interface de usuário (*user ISA*) do hardware é preservada, permitindo que as aplicações convidadas executem sem necessidade de modificações. Esse conceito é ilustrado na figura 4.17, adaptada [Ferre et al. 2006].

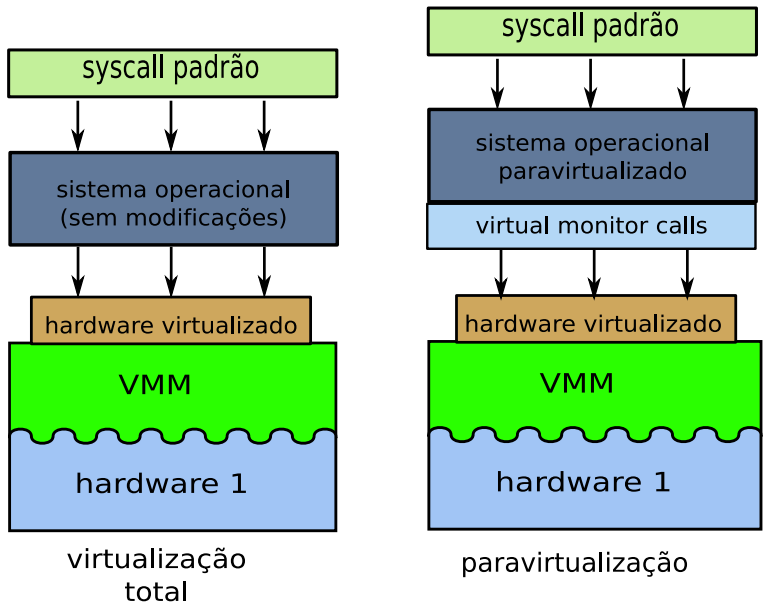

Figura 4.17. Relação entre a virtualização total e a paravirtualização.

Os primeiros ambientes a adotar a paravirtualização foram o *Denali* [Whitaker et al. 2002] e o *Xen* [Barham et al. 2003]. O *Denali* é um ambiente experimental de paravirtualização construído na Universidade de Washington, que pode suportar dezenas de milhares de máquinas virtuais sobre um computador *x86* convencional.

O projeto *Denali* não se preocupa em suportar sistemas operacionais comerciais, sendo voltado à execução maciça de minúsculas máquinas virtuais para serviços de rede. Já o ambiente de máquinas virtuais *Xen*, apresentado com mais detalhes na seção 4.3.3, permite executar sistemas operacionais convencionais como Linux e Windows, modificados para executar sobre seu hipervisor.

Embora exija que o sistema convidado seja adaptado ao hipervisor, o que diminui sua portabilidade, a paravirtualização permite que o sistema convidado acesse alguns recursos do hardware diretamente, sem a intermediação ativa do hipervisor. Nesses casos, o acesso ao hardware é apenas monitorado pelo hipervisor, que informa ao sistema convidado seus limites, como as áreas de memória e de disco disponíveis. O acesso aos demais dispositivos, como mouse e teclado, também é direto: o hipervisor apenas gerencia a ordem de acessos, no caso de múltiplos sistemas convidados em execução simultânea.

A diferença entre a virtualização total e a paravirtualização pode ser melhor compreendida observando-se a virtualização do acesso à memória. Na virtualização total, o hipervisor reserva um espaço de memória separado para cada sistema convidado; no entanto, todos os sistemas convidados "vêem" suas respectivas áreas de memória iniciando no endereço 0000_H. A figura 4.18 ilustra essa situação. Dessa forma, cada vez que um sistema convidado acessa a memória, o hipervisor traduz os endereços gerados por ele para as posições reais da área de memória daquele sistema convidado. Por outro lado, na paravirtualização, o hipervisor informa ao sistema operacional convidado as áreas de memória que este pode utilizar. Dessa forma, o sistema convidado pode acessar e gerenciar diretamente a memória reservada a ele, sem a interferência direta do hipervisor.

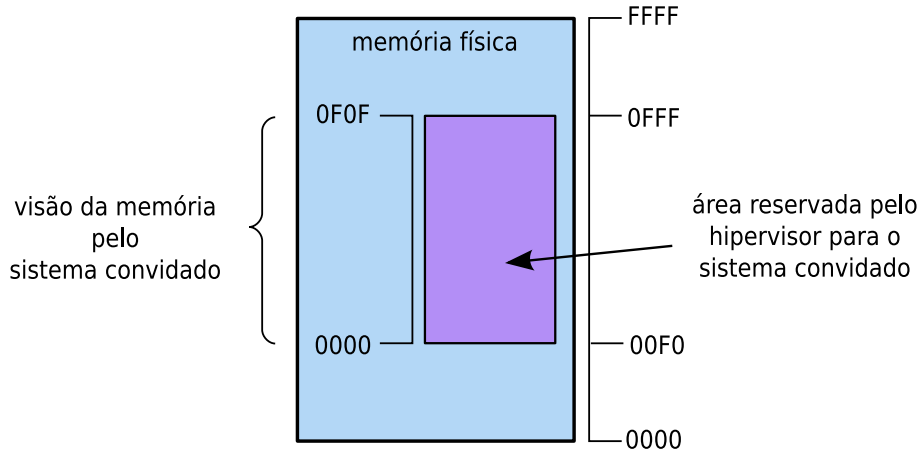

Figura 4.18. Visão da memória por um sistema convidado.

Apesar de exigir modificações nos sistemas operacionais convidados, a paravirtualização tem tido sucesso, por conta do desempenho obtido nos sistemas virtualizados, além de simplificar a interface de baixo nível dos sistemas convidados.

4.1.5.4. Melhoria de desempenho

De acordo com os princípios de Goldberg e Popek, o hipervisor deve permitir que a máquina virtual execute diretamente sobre o hardware sempre que possível, para não prejudicar o desempenho dos sistemas convidados. O hipervisor deve retomar o controle do processador somente quando a máquina virtual tentar executar operações que possam afetar o correto funcionamento do sistema, o conjunto de operações de outras máquinas virtuais ou do próprio hardware. O hipervisor deve então simular com segurança a operação solicitada e devolver o controle à máquina virtual.

Na prática, os hipervisores nativos e convidados raramente são usados em sua forma conceitual. Várias otimizações são inseridas nas arquiteturas apresentadas, com o objetivo principal de melhorar o desempenho das aplicações nos sistemas convidados. Como os pontos cruciais do desempenho dos sistemas de máquinas virtuais são as operações de entrada/saída, as principais otimizações utilizadas em sistemas de produção dizem respeito a essas operações. Quatro formas de otimização são usuais:

- Em hipervisores nativos (figura 4.19):

    1. O sistema convidado (*guest system*) acessa diretamente o hardware. Essa forma de acesso é implementada por modificações no núcleo do sistema convidado e no hipervisor. Essa otimização é implementada, por exemplo, no subsistema de gerência de memória do ambiente Xen [Barham et al. 2003].

- Em hipervisores convidados (figura 4.19):

    1. O sistema convidado (*guest system*) acessa diretamente o sistema nativo (*host system*). Essa otimização é implementada pelo hipervisor, oferecendo partes da API do sistema nativo ao sistema convidado. Um exemplo dessa otimização é a implementação do sistema de arquivos no *VMware* [VMware 2000]: em vez de reconstruir integralmente o sistema de arquivos sobre um dispositivo virtual provido pelo hipervisor, o sistema convidado faz uso da implementação de sistema de arquivos existente no sistema nativo.
    2. O sistema convidado (*guest system*) acessa diretamente o hardware. Essa otimização é implementada parcialmente pelo hipervisor e parcialmente pelo sistema nativo, pelo uso de um *device driver* específico. Um exemplo típico dessa otimização é o acesso direto a dispositivos físicos como leitor de CDs, hardware gráfico e interface de rede provida pelo sistema *VMware* aos sistemas operacionais convidados [VMware 2000].
    3. O hipervisor acessa diretamente o hardware. Neste caso, um *device driver* específico é instalado no sistema nativo, oferecendo ao hipervisor uma interface de baixo nível para acesso ao hardware subjacente. Essa abordagem, ilustrada na figura 4.20, é usada pelo sistema *VMware* [VMware 2000].

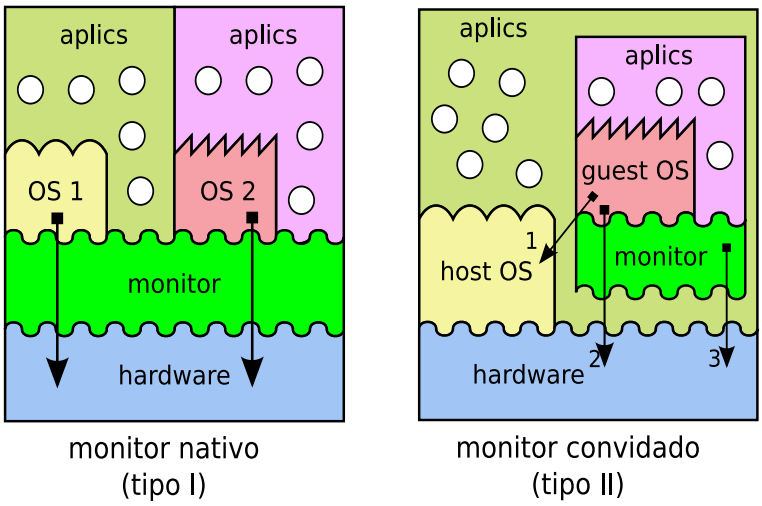

Figura 4.19. Otimizações em sistemas de máquinas virtuais.

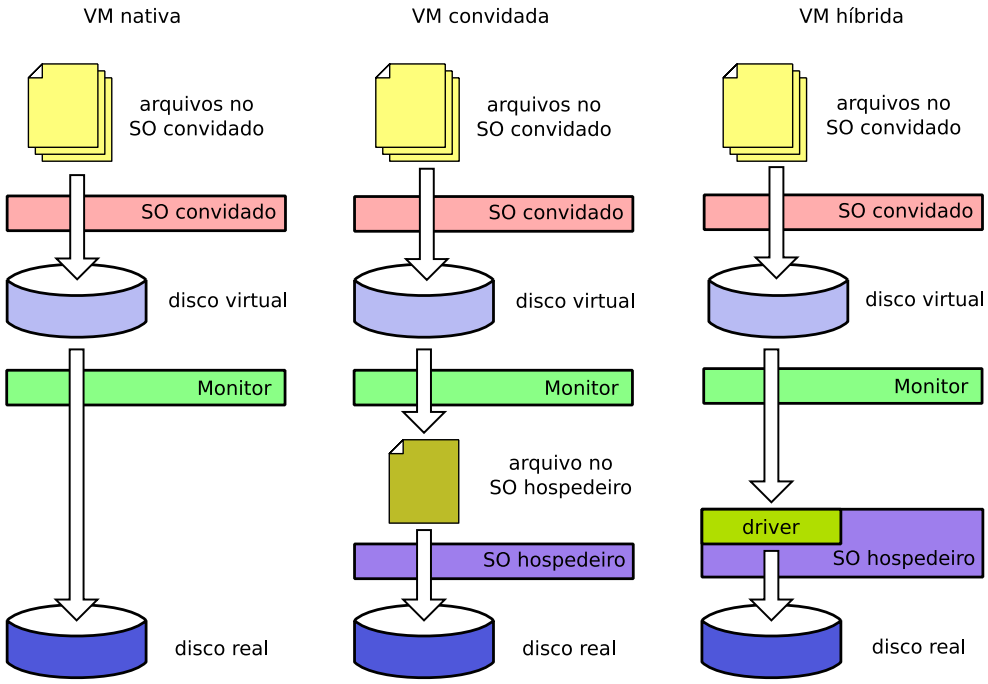

Figura 4.20. Desempenho de hipervisores nativos e convidados.

4.2. Virtualização e segurança

A evolução da tecnologia de máquinas virtuais tem permitido sua ampla adoção em sistemas de produção. Vários aspectos da construção de hipervisores, em sua maioria relacionados com o desempenho de execução dos sistemas convidados, foram resolvidos nos últimos anos [Rosenblum and Garfinkel 2005]. Atualmente, a principal utilização de máquinas virtuais no meio corporativo tem sido a consolidação de servidores, buscando a redução de custos em hardware, software e gerência do parque tecnológico [Newman et al. 2005, Ferre et al. 2006]. No entanto, vários trabalhos de pesquisa e desenvolvimento nos últimos anos comprovaram a eficácia da utilização de máquinas vir-

tuais no campo da segurança de sistemas. Esta seção discute como as propriedades dos hipervisores (apresentadas na seção 4.1.3.2) podem ser aplicadas na segurança de sistemas e discute alguns usos típicos de máquinas virtuais nesse contexto.

4.2.1. Aplicações da virtualização em segurança

Conforme [Krause and Tipton 1999], são três os princípios básicos para garantir a segurança da informação:

- *Confidencialidade*: a informação somente está visível a sujeitos (usuários e/ou processos) explicitamente autorizados;
- *Disponibilidade*: a informação deve estar prontamente disponível sempre que for necessária;
- *Integridade*: a informação somente pode ser modificada por sujeitos explicitamente autorizados e de formas claramente definidas.

Além destes, outros critérios devem ser respeitados para um sistema ser considerado seguro [Sêmola 2003]:

- *Autenticidade*: garante que a informação ou o usuário da mesma é autêntico, ou seja, garante que a entidade envolvida é quem afirma ser;
- *Não-repúdio*: não é possível negar a existência ou autoria de uma operação que criou, modificou ou destruiu uma informação;
- *Auditoria*: implica no registro das ações realizadas no sistema, identificando os sujeitos e recursos envolvidos, as operações realizadas, seus horários, locais e outros dados relevantes.

Algumas das propriedades conceituais da virtualização discutidas na seção 4.1.3.2 podem ser úteis para o atendimento desses critérios de segurança:

- *Isolamento*: ao manter os ambientes virtuais isolados entre si e do sistema real subjacente, o hipervisor provê a confidencialidade de dados entre os sistemas convidados. Adicionalmente, como os dados presentes em uma máquina virtual só podem ser acessados pelas respectivas aplicações convidadas, sua integridade é preservada. Além disso, o isolamento permite a contenção de erros de software acidentais ou intencionais no âmbito da máquina virtual [LeVasseur et al. 2004, Tan et al. 2007], o que permite melhorar a disponibilidade dos sistemas;
- *Controle de recursos*: Como o hipervisor intermedeia os acessos do sistema convidado ao hardware, é possível implementar mecanismos para verificar a consistência desses acessos e de seus resultados, aumentando a integridade do sistema convidado; da mesma forma, é possível acompanhar e registrar as atividades do sistema convidado, para fins de auditoria [Dunlap et al. 2002];

- *Inspeção*: a visão privilegiada do hipervisor sobre o estado interno do sistema convidado permite extrair informações deste para o sistema hospedeiro, permitindo implementar externamente mecanismos de verificação de integridade do ambiente convidado, como antivírus e detectores de intrusão [Laureano et al. 2007]; além disso, a capacidade de inspeção do sistema convidado, aliada ao isolamento provido pelo hipervisor, torna as máquinas virtuais excelentes "balões de ensaio" para o estudo de aplicações maliciosas como vírus e *trojans*;

- *Encapsulamento*: a possibilidade de salvar/restaurar o estado do sistema convidado torna viável a implementação de mecanismos de *rollback* úteis no caso de quebra da integridade do sistema convidado; da mesma forma, a migração de máquinas virtuais é uma solução viável para o problema da disponibilidade [Fu and Xu 2005];

Nos últimos anos, muitos projetos de pesquisa estudaram a aplicação das propriedades das máquinas virtuais na construção de sistemas computacionais seguros e confiáveis. Algumas dessas pesquisas visam proteger e aumentar a disponibilidade de sistemas e serviços, outras visam estudar o comportamento de aplicações maliciosas, mas também há estudos visando criar aplicações maliciosas que tirem proveito da virtualização, ao menos como prova de conceito. Esses trabalhos abordam diversas aplicações, como o confinamento de serviços, a detecção de intrusão, a análise de *malwares*, a construção de *honeypots* e *rootkits*, técnicas de contingenciamento e de tolerância a falhas. Algumas dessas pesquisas serão discutidas nas próximas seções.

4.2.2. Confinamento de aplicações

Em um sistema operacional, cada processo tem acesso a um conjunto de recursos para funcionar. Esse conjunto é definido através de regras de controle de acesso impostas pelo núcleo do sistema. Além disso, a lógica interna do processo também pode limitar os recursos acessíveis a seu respectivo usuário. Por exemplo, um servidor Web pode acessar os arquivos autorizados pelas permissões do sistema de arquivos, mas também possui regras internas para limitar os arquivos e diretórios acessíveis aos clientes. Todavia, serviços mal configurados, erros nas regras de controle de acesso, vulnerabilidades no serviço ou no sistema operacional podem expor informações indevidamente, comprometendo a confidencialidade e integridade do sistema. Esse problema pode ser atenuado de diversas formas:

- Monitorar a aplicação usando IDS, antivírus etc. Esta solução pode requerer um trabalho de configuração significativo; além disso, essas ferramentas também são processos, consumindo recursos e sendo passíveis de falhas ou subversão;

- Designar um equipamento separado para a aplicação sensível executar isoladamente, o que pode custar caro;

- Colocar a aplicação para executar dentro de uma máquina virtual, isolando-a do restante do sistema hospedeiro.

Em princípio, qualquer hipervisor convencional pode ser usado para confinar uma aplicação. Todavia, neste caso específico, a principal motivação para o uso de máquinas virtuais é a propriedade de isolamento (seção 4.1.3.2). Como essa propriedade não

implica necessariamente a virtualização do processador ou dos demais recursos de hardware, técnicas simplificadas e com baixo custo computacional foram concebidas para implementá-la. Essas técnicas são conhecidas como *servidores virtuais*.

Em um servidor virtual, a interface binária (ABI), composta pelas chamadas de sistema e instruções do processador, é totalmente preservada. O espaço de usuário do sistema operacional é dividido em áreas isoladas denominadas *domínios* virtuais. Cada domínio virtual recebe uma parcela dos recursos do sistema, como memória, tempo de processador e espaço em disco. Alguns recursos do sistema real podem ser virtualizados, como as interfaces de rede: cada domínio tem sua própria interface virtual e seu próprio endereço de rede. Em algumas implementações, cada domínio virtual pode impor seu próprio espaço de nomes: assim, pode-se ter um usuário `pedro` no domínio d_3 e outro usuário `pedro` no domínio d_7, sem conflitos. A distinção de espaços de nomes pode se estender a outros recursos do sistema: identificadores de processos, semáforos, árvores de diretórios, etc.

Os processos confinados em um determinado domínio podem interagir entre si, criar novos processos e usar os recursos presentes naquele domínio, respeitando as regras de controle de acesso associadas a esses recursos. Todavia, processos em um domínio não podem interagir com processos em outro domínio, não podem mudar de domínio, criar processos em outros domínios, nem usar recursos de outros domínios. Para cada domínio, os demais domínios são máquinas distintas. Normalmente é definido um domínio d_0, ou *domínio de gerência*, cujos processos têm acesso aos demais domínios. A figura 4.21 mostra a estrutura típica de um sistema de servidores virtuais. Nela, pode-se observar que um processo pode migrar de d_0 para d_1, mas que processos em d_1 não podem migrar para outros domínios. A comunicação entre processos confinados em domínios distintos (d_2 e d_3) também é proibida.

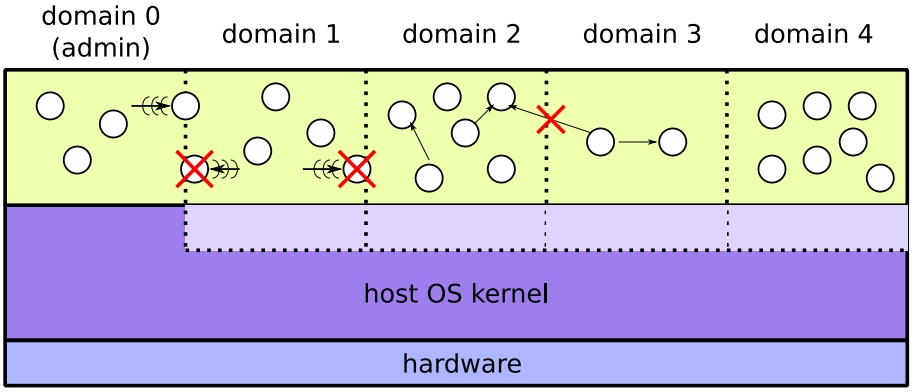

Figura 4.21. Servidores virtuais.

Há várias implementações disponíveis de mecanismos de confinamento de processos. A técnica mais antiga é realizada através da chamada de sistema `chroot`. O Sistema *FreeBSD* implementa o ambiente *Jails* [Kamp and Watson 2000] para este fim. Existem implementações similares em outros sistemas operacionais, como as *Zones* do sistema *Solaris* e os sistemas *Virtuozzo/OpenVZ* e *Vservers/FreeVPS*, para Linux.

4.2.3. Detecção de intrusão

Sistemas de detecção de intrusão (IDS – *Intrusion Detection Systems*) têm como função monitorar uma rede ou sistema computacional, buscando detectar ataques ou atividades maliciosas. Esses sistemas continuamente coletam dados do sistema e os analisam através de técnicas diversas, como reconhecimento de padrões, análise estatística e mineração de dados. Ao detectar uma atividade maliciosa, podem alertar o administrador do sistema e/ou ativar contra-medidas. De acordo com a origem da informação analisada, os IDSs podem ser classificados como:

- *IDSs de máquina* (HIDS – *Host-based IDS*): monitoram um computador para identificar atividades maliciosas locais, como subversão de serviços, vírus e *trojans*.
- *IDSs de rede* (NIDS – *Network-based IDS*): monitoram o tráfego de uma rede, para identificar ataques aos computadores a ela conectados.

Sistemas NIDS monitoram vários computadores simultaneamente, mas sua eficácia diminui na medida em que o tráfego da rede aumenta, pela necessidade de analisar pacotes mais rapidamente. Além disso, o uso de protocolos de rede cifrados torna o conteúdo dos pacotes opaco ao NIDS. Sistemas HIDS não sofrem desses problemas, pois analisam as informações disponíveis dentro de cada sistema. Todavia, um HIDS é um processo local, que pode ser desativado ou subvertido por um ataque bem-sucedido.

As máquinas virtuais podem ser úteis na proteção de sistemas HIDS. Através da propriedade de inspeção, o hipervisor pode extrair informações de um sistema convidado e encaminhá-las para análise por um detector de intrusão executando no sistema nativo, ou em outra máquina virtual. O isolamento entre máquinas virtuais assegura a integridade das informações coletadas e do próprio detector de intrusão. Por fim, a propriedade de controle de recursos permite ao hipervisor intervir no sistema convidado, para interromper atividades consideradas suspeitas pelo detector de intrusão.

No sistema *ReVirt* [Dunlap et al. 2002], uma camada construída entre o hipervisor e o sistema hospedeiro recebe as mensagens de *syslog*[2] geradas no sistema convidado e as registra no sistema hospedeiro. Essas mensagens são então usadas para detectar erros ou atividades maliciosas envolvendo os serviços do sistema convidado. Além disso, a camada *ReVirt* realiza *checkpoints* regulares do sistema convidado, que a permitem reverter o sistema a um estado anterior seguro, no caso de um ataque bem sucedido.

O trabalho [Garfinkel and Rosenblum 2003] descreve uma arquitetura para a detecção de intrusão em máquinas virtuais denominada VMI IDS (*Virtual Machine Introspection Intrusion Detection System*). Sua abordagem considera o uso de um hipervisor nativo, executando diretamente sobre o hardware. O IDS executa em uma das máquinas virtuais e observa dados obtidos das demais máquinas virtuais, buscando evidências de intrusão. Somente o estado global da máquina virtual (registradores, consumo de memória, etc) é analisado, sem levar em conta as atividades dos processos nela contidos, ou seja, o sistema proposto não tem a capacidade de avaliar processos individuais. Por isso, sua capacidade de resposta é limitada: caso haja suspeita de intrusão, a máquina virtual

[2] *Daemon* UNIX que registra as mensagens de *log* geradas pelas aplicações em execução no sistema.

comprometida é suspensa até que essa suspeita seja confirmada; nesse caso, a mesma pode ser reiniciada ou revertida a um estado anterior seguro (*rollback*).

Por outro lado, o trabalho [Laureano et al. 2004, Laureano et al. 2007] utiliza um hipervisor convidado modificado, que extrai informações do sistema convidado e as submete a um detector de intrusão executando no sistema hospedeiro. A detecção de intrusão é baseada nas seqüências de chamadas de sistema emitidas pelos processos do sistema convidado; dessa forma, cada processo convidado pode ser analisado separadamente. Caso o comportamento de um processo seja considerado anômalo, o hipervisor gera um alerta de suspeita e restringe as chamadas de sistema disponíveis ao processo.

4.2.4. Análise de programas maliciosos

[Klaus 1999, Skoudis and Zeltser 2003] consideram a existência de dois tipos de programas prejudiciais (figura 4.22):

- *Intencionais*: programas escritos para se infiltrar em um sistema, sem conhecimento de seus usuários, com a intenção de causar dano, furto ou seqüestro de informações. Esses programas são conhecidos como *malwares* (de *malicious softwares*);
- *Não-intencionais*: programas normais contendo erros de programação ou de configuração que permitam a manipulação não-autorizada das informações de um sistema; esses erros de programação ou de configuração são denominados *vulnerabilidades*.

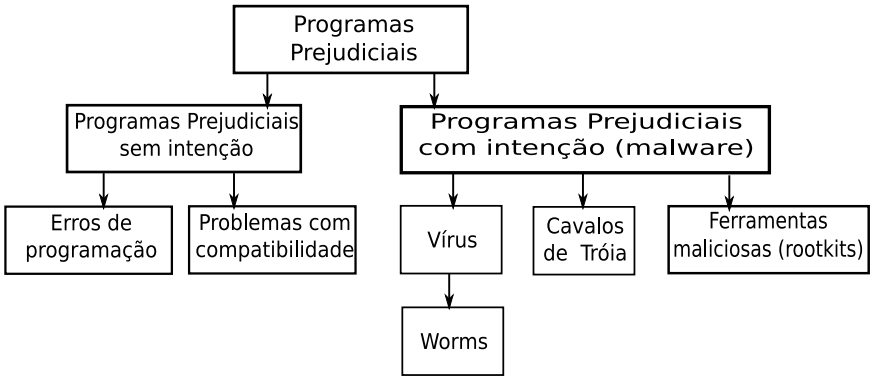

Figura 4.22. Classificação dos Programas Prejudiciais [Klaus 1999].

Para [Skoudis and Zeltser 2003], os malwares são conjuntos de instruções executadas em um computador e que fazem o sistema realizar algo que um atacante deseja. De forma geral, os malwares podem ser classificados em:

- *Vírus*: classe de software malicioso com a habilidade de se auto-replicar e infectar partes do sistema operacional ou dos programas de aplicação, visando causar a perda ou danos nos dados;
- *Worm*: designa qualquer software capaz de propagar a si próprio em uma rede. Habitualmente os *worms* invadem os sistemas computacionais através da exploração de falhas nos serviços de rede;

- *Cavalo de Tróia*: programa com função aparentemente útil, que também realiza ações escondidas visando roubar informações ou provocar danos no sistema;
- *Rootkit*: conjunto de ferramentas usadas por um invasor para ocultar sua presença em um sistema invadido. Os *rootkits* mais sofisticados envolvem modificações no próprio sistema operacional, para ocultar os recursos (como processos, arquivos e conexões de rede) usados pelo invasor.

A análise de um *malware* consiste em estudar o conteúdo (análise estática) e o comportamento (análise dinâmica) do programa, para descobrir seus objetivos, seu método de propagação, sua forma de dissimulação no sistema, encontrar evidências que possam indicar sua presença ou atividade e implementar formas de detectá-lo, removê-lo do sistema e impedir novas invasões. A análise estática se baseia no estudo do código binário do *malware*; ela é cada vez mais difícil de realizar, devido ao emprego de técnicas de dissimulação e cifragem [Beaucamps and Filiol 2007]. A análise dinâmica consiste na observação da execução do *malware* e de seus efeitos no sistema. Nesse contexto, o uso de uma máquina virtual é benéfico, pelas seguintes razões:

- Não é necessário dedicar uma máquina real "limpa" para cada análise;
- Torna-se simples salvar e restaurar estados da máquina virtual, permitindo desfazer os efeitos de uma intrusão; além disso, a comparação entre os estados antes de depois da intrusão permite compreender melhor seus efeitos no sistema;
- A verificação de informações de baixo nível (como o estado da memória, registradores, dados dentro do núcleo) torna-se mais simples, através da capacidade de inspeção do hipervisor;
- a tradução dinâmica de instruções pode ser usada para instrumentar o fluxo de instruções executado pelo *malware*.

Com base nessas constatações, diversas ferramentas e serviços de análise dinâmica automatizada de programas suspeitos têm sido propostos, como *Anubis*, *CWSandbox* e *Joebox* [Bayer et al. 2006, Willems et al. 2007]. Todavia, essa abordagem ainda não é perfeita. Vários estudos recentes têm demonstrado que as implementações atuais de hipervisores têm erros de projeto e/ou implementação que podem expor o sistema hospedeiro a ataques vindos do sistema convidado [Ormandy 2007], comprometendo a segurança do hipervisor e do próprio sistema hospedeiro. Além disso, a virtualização de plataformas correntes, como a de sistemas PC *Intel x86*, é bastante complexa, estando sujeita a problemas na virtualização de certos aspectos do hardware (conforme discutido na seção 4.1.3.3). Essas imperfeições abrem a porta para que *malwares* possam identificar o sistema invadido e comportar-se de forma distinta caso estejam em uma máquina virtual ou em um computador real [Raffetseder et al. 2007]. Com isso, a análise dinâmica do *malware* poderá não corresponder à sua execução em um sistema real. Por outro lado, outras pesquisas têm mostrado que há formas de tornar mais difícil a detecção de um sistema virtualizado por parte das aplicações convidadas [Liston and Skoudis 2006].

Em [Kwan and Durfee 2007] é proposta uma estrutura utilizando máquinas virtuais para verificação da integridade de um sistema. Neste sistema, chamado *Vault*, um agente de verificação é instalado nos hipervisores em uso. Estes agentes realizam a troca de mensagens visando garantir a segurança do sistema convidado monitorado. O principio do projeto é garantir que os sistemas monitorados não sejam subvertidos por *malwares* convencionais.

4.2.5. Honeypots e honeynets

Um *honeypot* é um sistema explicitamente preparado para ser atacado e invadido. Os *honeypots* podem ser vistos como "armadilhas", pois servem para atrair e estudar o tráfego de rede malicioso, como *worms* e ataques humanos. Como o *honeypot* não corresponde a serviços legítimos da rede e pode ser facilmente atacado, serve como alerta para a presença de atacantes na rede. Além disso, ele desvia a atenção dos atacantes dos servidores reais do sistema.

Os *honeypots* podem ser de *baixa* ou de *alta interatividade*. Nos *honeypots* de baixa interatividade, o atacante interage com emulações de serviços de rede, que não correspondem a serviços reais e portanto não serão realmente comprometidos. Um exemplo dessa abordagem é o *Honeyd*, um *daemon* UNIX que emula várias pilhas e serviços de rede [Provos 2004]. Já os *honeypots* de alta interatividade expõem sistemas e serviços reais – que podem ser comprometidos – aos atacantes. *Honeypots* de alta interatividade podem ser perigosos se mal configurados, pois podem ser invadidos e usados como plataformas para ataques ao restante da rede. Todavia, como os *honeypots* de alta interatividade são sistemas reais, as observações fornecidas por eles são mais detalhadas e correspondem melhor à realidade.

Uma *honeynet* [Honeynet Project 2001] é uma coleção de *honeypots* de alta interatividade com diferentes sistemas operacionais, configurações e serviços de rede. Por sua diversidade, essa rede tem um alto potencial de atração de ataques. Além dos *honeypots*, uma *honeynet* possui *firewalls* para evitar que tráfego malicioso se propague a partir dela para outras redes, detectores de intrusão para monitorar as atividades maliciosas e servidores de *logging* para o registro de eventos. A implantação de uma *honeynet* implica em alocar vários computadores, com sistemas operacionais e serviços distintos, o que pode significar um alto custo de implantação e operação. Nesse contexto, máquinas virtuais podem ser usadas para construir *honeynets virtuais*.

Uma *honeynet virtual* [Honeynet Project 2003] é simplesmente uma *honeynet* construída com máquinas virtuais ao invés de computadores reais. A virtualização traz vantagens, como o menor custo de implantação e gerência, mas pode também trazer inconvenientes, como limitar as possibilidades de *honeypots* aos sistemas que podem ser virtualizados sobre a plataforma computacional escolhida. Além disso, caso o sistema nativo seja comprometido, toda a *honeynet* estará comprometida. Para [Honeynet Project 2003], uma *honeynet* pode ser *totalmente virtual*, quando todos os computadores envolvidos são máquinas virtuais, ou *híbrida*, quando é composta por *honeypots* virtuais e máquinas reais para as funções de *firewall*, detecção de intrusão e *logging*.

4.2.6. *Rootkits*

Conforme apresentado na seção 4.2.4, um *rootkit* é uma ferramenta que visa ocultar a presença de um intruso no sistema. Os primeiros *rootkits* consistiam em substituir alguns comandos do sistema (como `ls`, `ps` e `netstat`) por versões com a mesma funcionalidade, mas que escondiam os arquivos, processos e conexões de rede do intruso, tornando-o "invisível". Posteriormente, foram desenvolvidos *rootkits* mais elaborados, que substituem bibliotecas do sistema com a mesma finalidade. Os *kernel rootkits* consistem em módulos de núcleo que alteram a implementação das chamadas de sistema dentro do núcleo, permitindo esconder o intruso de forma ainda mais profunda no sistema [Kruegel et al. 2004].

Recentemente, os avanços no suporte de hardware à virtualização possibilitaram o desenvolvimento de *rootkits* baseado em máquinas virtuais, ou *VM-based rootkits* (VMBR) [King and Chen 2006]. O funcionamento de um VMBR é conceitualmente simples: ao ser instalado, ele se torna um hipervisor, virtualizando todo o hardware da máquina e transformando o sistema operacional invadido em um sistema convidado (conforme apresentado na figura 4.23). Assim, toda a funcionalidade maliciosa do *rootkit* se encontra abaixo do sistema operacional (e fora do alcance dele), sendo teoricamente indetectável por detectores de *rootkits* convencionais.

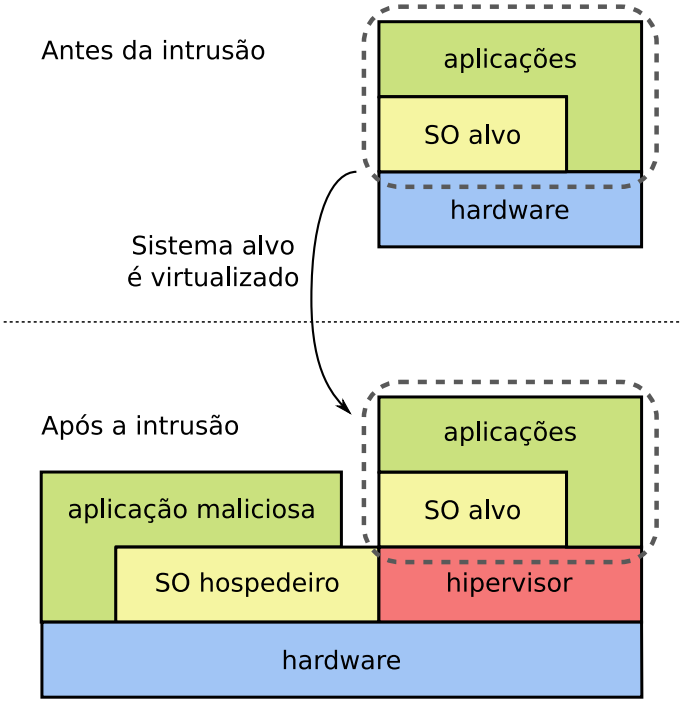

Figura 4.23. Sistema antes e depois de ser afetado por um VMBR [King and Chen 2006].

Os *rootkits* baseados em virtualização mais conhecidos são o *Subvirt* (que é uma prova de conceito) [King and Chen 2006], o *Vitriol* (que usa a tecnologia *VT-x* dos processadores Intel) [Zovi 2006] e o *BluePill* (que usa a tecnologia equivalente *SVM*, da AMD) [Rutkowska 2006]. Embora os processadores sejam diferentes, o princípio de funcionamento do suporte de hardware à virtualização é análogo em ambas as plataformas.

Da mesma forma que um hipervisor nativo, estes *rootkits* modificam a seqüência de inicialização da máquina afetada, carregando primeiro o seu código e posteriormente o sistema operacional original, em um ambiente convidado. Quando ativo, o VMBR torna-se incontornável: ele pode interceptar todos os acessos ao hardware e inspecionar todo o estado interno do sistema operacional. Embora este tipo de *rootkit* seja teoricamente indetectável, vários pesquisadores afirmam ser possível detectar anomalias em um sistema virtualizado, observado o comportamento dos processos, a alocação de recursos, diferenças de temporização nos acessos ao hardware, e falhas no suporte à virtualização, que permitiriam detectar a presença desses *rootkits* [Rutkowska 2004, Garfinkel et al. 2007].

4.2.7. Consolidação de servidores, planos de contingência e migração

Uma das aplicações mais freqüentes da virtualização é a consolidação de servidores, que consiste em transferir serviços em máquinas físicas para máquinas virtuais, para diminuir o número de equipamentos da organização. Busca-se com isso aumentar a produtividade da infra-estrutura, simplificar a gerência do ambiente, aumentar a segurança, diminuir o consumo de energia e economizar recursos humanos, físicos e financeiros. Com a virtualização, os recursos físicos podem ser dinamicamente alocados aos sistemas operacionais de acordo com suas necessidades, proporcionando balanceamento de carga dinâmico e um melhor controle de uso dos recursos dos servidores [Newman et al. 2005, Ferre et al. 2006]. Além disso, a redução do número de equipamentos permite mais investimento em tecnologias de melhoria da disponibilidade, como fontes de alimentação redundantes e controladores de disco com suporte a *hot-swapping*.

Um plano de contingência visa assegurar a disponibilidade de sistemas críticos e facilitar a continuidade das operações durante uma crise. Esse plano é aplicado após algum incidente nos sistemas de computação. As máquinas virtuais podem desempenhar um papel importante nos planos de contingência de servidores e serviços, a um custo de hardware e software relativamente baixo [Newman et al. 2005, Ferre et al. 2006]. Como máquinas virtuais podem ser salvas na forma de arquivos, *checkpoints* dos servidores críticos podem ser salvos em equipamentos distintos. Assim, quando um serviço falhar, uma cópia atualizada do mesmo pode ser rapidamente colocada em funcionamento.

Hipervisores modernos implementam facilidades de *migração* de máquinas virtuais. Na migração, uma máquina virtual e seu sistema convidado são transferidos de um hipervisor para outro, executando em equipamentos distintos. A máquina virtual tem seu estado preservado e prossegue sua execução no hipervisor de destino assim que a migração é concluída. De acordo com [Clark et al. 2005], as técnicas mais freqüentes para implementar a migração são:

- *stop-and-copy*: consiste em suspender a máquina virtual, transferir o conteúdo de sua memória para o hipervisor de destino e retomar a execução em seguida. É uma abordagem simples, mas implica em parar completamente os serviços oferecidos pelo sistema convidado enquanto durar a migração (que pode demorar algumas dezenas de segundos);

- *demand-migration*: a máquina virtual é suspensa apenas durante a cópia das estruturas de memória do núcleo para o hipervisor de destino, o que dura alguns

milissegundos. Em seguida, a execução da máquina virtual é retomada e o restante das páginas de memória da máquina virtual é transferido sob demanda, através dos mecanismos de tratamento de faltas de página. Nesta abordagem a interrupção do serviço tem duração mínima, mas a migração completa pode demorar muito tempo.

- *pre-copy*: consiste basicamente em copiar para o hipervisor de destino todas as páginas de memória da máquina virtual enquanto esta executa; a seguir, a máquina virtual é suspensa e as páginas modificadas depois da cópia inicial são novamente copiadas no destino; uma vez terminada a cópia dessas páginas, a máquina pode retomar sua execução no destino. Esta abordagem, usada no hipervisor *Xen* [Barham et al. 2003], é a que oferece o melhor compromisso entre o tempo de suspensão do serviço e a duração total da migração.

A migração de máquinas virtuais é uma solução útil para a continuidade de serviços no caso de manutenções programadas, mas ela também pode ser usada nos casos de falhas não-previstas, como demonstra o trabalho [Fu and Xu 2005].

4.2.8. Tolerância a faltas

A propriedade de isolamento das máquinas virtuais pode ser usada para restringir a propagação de erros de software nos sistemas. Alguns projetos de pesquisa exploram a possibilidade de usar máquinas virtuais para encapsular partes de sistemas operacionais ou de aplicativos, buscando aumentar assim a sua disponibilidade. Por exemplo, a maioria das falhas em um sistema operacional é causada por *drivers* construídos por terceiros [Herder et al. 2006]. Como estes *drivers* fazem parte do sistema operacional, eles executam no nível mais privilegiado do processador. Baseado na virtualização do processador é possível criar um ambiente isolado para a execução de *drivers* suspeitos ou instáveis, restringindo o alcance de eventuais erros nos mesmos.

O projeto *iKernel* (*isolation Kernel*) [Tan et al. 2007] utiliza essa abordagem. Nesse projeto, um núcleo Linux é executado como um sistema convidado. A execução de *drivers* é realizada no sistema convidado, e o resultado da execução é devolvido ao sistema hospedeiro. A comunicação entre o sistema convidado e o hospedeiro é realizada através de uma área de memória compartilhada entre ambos os sistemas, estritamente isolada do restante da memória. Eventuais erros nos *drivers* ficam restritos ao sistema convidado, que pode ser facilmente reiniciado, não afetando o hospedeiro. Testes realizados utilizando o hipervisor *QEMU* (seção 4.3.5) comprovaram a viabilidade da proposta. Todavia, alguns *drivers* de dispositivos não são facilmente isoláveis, como os *drivers* gráficos.

Outro projeto que usa a mesma abordagem é o *L4Ka* [LeVasseur et al. 2004]. A principal diferença entre este e o *iKernel* diz respeito ao nível de virtualização do sistema convidado, que neste caso não se beneficia do suporte de hardware à virtualização. Neste projeto, alguns *drivers* são executado em um sistema convidado com acesso somente aos recursos necessários para a sua execução.

4.3. Exemplos de máquinas virtuais

Esta seção apresenta alguns sistemas de máquinas virtuais de uso corrente. Serão apresentados os sistemas *VMWare*, *FreeBSD Jails*, *Xen*, *User-Mode Linux*, *QEMU*, *Valgrind* e *JVM*. Entre eles há máquinas virtuais de aplicação e de sistema, com virtualização total ou paravirtualização, além de abordagens hibridas. Eles foram escolhidos por estarem entre os mais representativos de suas respectivas classes.

4.3.1. VMware

Atualmente, o *VMware* é a máquina virtual para a plataforma x86 de uso mais difundido, provendo uma implementação completa da interface x86 ao sistema convidado. Embora essa interface seja extremamente genérica para o sistema convidado, acaba conduzindo a um hipervisor mais complexo. Como podem existir vários sistemas operacionais em execução sobre mesmo hardware, o hipervisor tem que emular certas instruções para representar corretamente um processador virtual em cada máquina virtual, fazendo uso intensivo dos mecanismos de tradução dinâmica [VMware 2000, Newman et al. 2005]. Atualmente, a *VMware* produz vários produtos com hipervisores nativos e convidados:

- Hipervisor convidado:
  - *VMware Workstation*: primeira versão comercial da máquina virtual, lançada em 1999, para ambientes *desktop*;
  - *VMware Fusion*: versão experimental para o sistema operacional *Mac OS* com processadores Intel;
  - *VMware Player*: versão gratuita do *VMware Workstation*, com as mesmas funcionalidades mas limitado a executar máquinas virtuais criadas previamente com versões comerciais;
  - *VMWare Server*: conta com vários recursos do *VMware Workstation*, mas é voltado para pequenas e médias empresas;
- Hipervisor nativo:
  - *VMware ESX Server*: para servidores de grande porte, possui um núcleo proprietário chamado *vmkernel* e Utiliza o *Red Hat Linux* para prover outros serviços, tais como a gerência de usuários.

O *VMware Workstation* utiliza as estratégias de virtualização total e tradução dinâmica (seção 4.1.5). O *VMware ESX Server* implementa ainda a paravirtualização. Por razões de desempenho, o hipervisor do *VMware* utiliza uma abordagem híbrida (seção 4.1.5.4) para implementar a interface do hipervisor com as máquinas virtuais [Sugerman et al. 2001]. O controle de exceção e o gerenciamento de memória são realizados por acesso direto ao hardware, mas o controle de entrada/saída usa o sistema hospedeiro. Para garantir que não ocorra nenhuma colisão de memória entre o sistema convidado e o real, o hipervisor *VMware* aloca uma parte da memória para uso exclusivo de cada sistema convidado.

Para controlar o sistema convidado, o *VMware Workstation* intercepta todas as interrupções do sistema convidado. Sempre que uma exceção é causada no convidado, é examinada primeiro pelo hipervisor. As interrupções de entrada/saída são remetidas para o sistema hospedeiro, para que sejam processadas corretamente. As exceções geradas pelas aplicações no sistema convidado (como as chamadas de sistema, por exemplo) são remetidas para o sistema convidado.

4.3.2. FreeBSD Jails

O sistema operacional *FreeBSD* oferece um mecanismo de confinamento de processos denominado *Jails*, criado para aumentar a segurança de serviços de rede. Esse mecanismo consiste em criar domínios de execução distintos (denominados *jails* ou celas), conforme descrito na seção 4.2.2. Cada cela contém um subconjunto de processos e recursos (arquivos, conexões de rede) que pode ser gerenciado de forma autônoma, como se fosse um sistema separado [Kamp and Watson 2000].

Cada domínio é criado a partir de um diretório previamente preparado no sistema de arquivos. Um processo que executa a chamada de sistema `jail` cria uma nova cela e é colocado dentro dela, de onde não pode mais sair, nem seus filhos. Além disso, os processos em um domínio não podem:

- Reconfigurar o núcleo (através da chamada `sysctl`, por exemplo);
- Carregar/retirar módulos do núcleo;
- Mudar configurações de rede (interfaces e rotas);
- Montar/desmontar sistemas de arquivos;
- Criar novos *devices*;
- Realizar modificações de configurações do núcleo em tempo de execução;
- Acessar recursos que não pertençam ao seu próprio domínio.

Essas restrições se aplicam mesmo a processos que estejam executando com privilégios de administrador (*root*).

Pode-se considerar que o sistema *FreeBSD Jails* virtualiza somente partes do sistema hospedeiro, como a árvore de diretórios (cada domínio tem sua própria visão do sistema de arquivos), espaços de nomes (cada domínio mantém seus próprios identificadores de usuários, processos e recursos de IPC) e interfaces de rede (cada domínio tem sua interface virtual, com endereço IP próprio). Os demais recursos (como as instruções de máquina e chamadas de sistema) são preservadas, ou melhor, podem ser usadas diretamente. Essa virtualização parcial demanda um custo computacional muito baixo, mas exige que todos os sistemas convidados executem sobre o mesmo núcleo.

4.3.3. Xen

O ambiente *Xen* é um hipervisor nativo para a plataforma *x86* que implementa a paravirtualização. Ele permite executar sistemas operacionais como Linux e Windows especialmente modificados para executar sobre o hipervisor [Barham et al. 2003]. Versões mais recentes do sistema *Xen* utilizam o suporte de virtualização disponível nos processadores atuais, o que torna possível a execução de sistemas operacionais convidados sem modificações, embora com um desempenho ligeiramente menor que no caso de sistemas paravirtualizados. Conforme seus criadores, o custo e impacto das alterações nos sistemas convidados são baixos e a diminuição do custo da virtualização compensa essas alterações (a degradação média de desempenho observada em sistemas virtualizados sobre a plataforma *Xen* não excede 5%). As principais modificações impostas pelo ambiente *Xen* a um sistema operacional convidado são:

- O mecanismo de entrega de interrupções passa a usar um serviço de eventos oferecido pelo hipervisor; o núcleo convidado deve registrar um vetor de tratadores de exceções junto ao hipervisor;
- as operações de entrada/saída de dispositivos são feitas através de uma interface simplificada, independente de dispositivo, que usa *buffers* circulares de tipo produtor/consumidor;
- o núcleo convidado pode consultar diretamente as tabelas de segmentos e páginas da memória usada por ele e por suas aplicações, mas as modificações nas tabelas devem ser solicitadas ao hipervisor;
- o núcleo convidado deve executar em um nível de privilégio inferior ao do hipervisor;
- o núcleo convidado deve implementar uma função de tratamento das chamadas de sistema de suas aplicações, para evitar que elas tenham de passar pelo hipervisor antes de chegar ao núcleo convidado.

Como o hipervisor deve acessar os dispositivos de hardware, ele deve dispor dos *drivers* adequados. Já os núcleos convidados não precisam de *drivers* específicos, pois eles acessam dispositivos virtuais através de uma interface simplificada. Para evitar o desenvolvimento de *drivers* específicos para o hipervisor, o ambiente *Xen* usa uma abordagem alternativa: a primeira máquina virtual (chamada VM_0) pode acessar o hardware diretamente e provê os *drivers* necessários ao hipervisor. As demais máquinas virtuais ($VM_i, i > 0$) acessam o hardware virtual através do hipervisor, que usa os *drivers* da máquina VM_0 conforme necessário. Essa abordagem, apresentada na figura 4.24, simplifica muito a evolução do hipervisor, por permitir utilizar os *drivers* desenvolvidos para o sistema Linux.

O hipervisor *Xen* pode ser considerado uma tecnologia madura, sendo muito utilizado em sistemas de produção. O seu código-fonte está liberado sob a licença *GNU General Public Licence* (GPL). Atualmente, o ambiente *Xen* suporta os sistemas Windows, Linux e NetBSD. Várias distribuições Linux já possuem suporte nativo ao *Xen*.

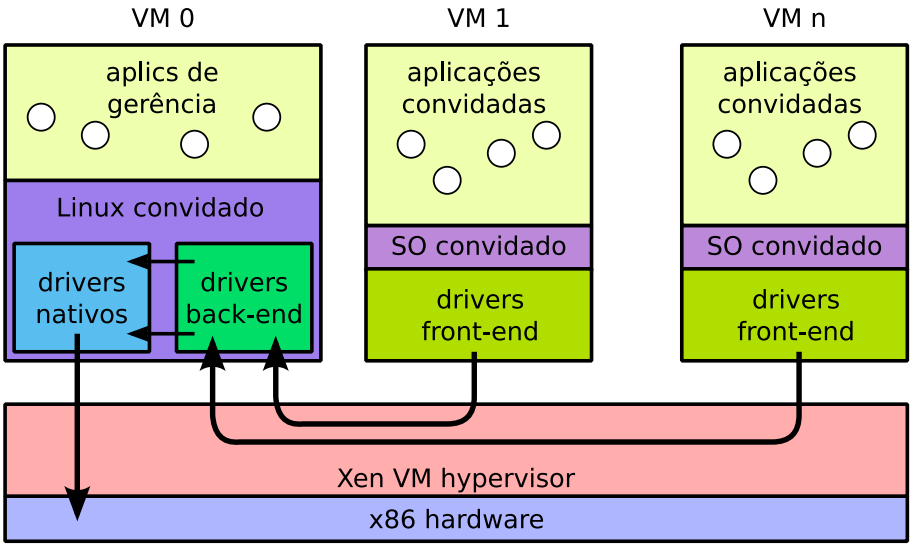

Figura 4.24. O hipervisor *Xen*.

4.3.4. User-Mode Linux

O *User-Mode Linux* foi proposto por Jeff Dike em 2000, como uma alternativa de uso de máquinas virtuais no ambiente Linux [Dike 2000]. O núcleo do Linux foi portado de forma a poder executar sobre si mesmo, como um processo do próprio Linux. O resultado é um *user space* separado e isolado na forma de uma máquina virtual, que utiliza dispositivos de hardware virtualizados a partir dos serviços providos pelo sistema hospedeiro. Essa máquina virtual é capaz de executar todos os serviços e aplicações disponíveis para o sistema hospedeiro. Além disso, o custo de processamento e de memória das máquinas virtuais *User-Mode Linux* é geralmente menor que aquele imposto por outros hipervisores mais complexos.

O *User-Mode Linux* é hipervisor convidado, ou seja, executa na forma de um processo no sistema hospedeiro. Os processos em execução na máquina virtual não têm acesso direto aos recursos do sistema hospedeiro. A maior dificuldade na implementação do *User-Mode Linux* foi encontrar formas de virtualizar as funcionalidades do hardware para as chamadas de sistema do Linux, sobretudo a distinção entre o modo privilegiado do núcleo e o modo não-privilegiado de usuário. Um código somente pode estar em modo privilegiado se é confiável o suficiente para ter pleno acesso ao hardware, como o próprio núcleo do sistema operacional. O *User-Mode Linux* deve possuir uma distinção de privilégios equivalente para permitir que o seu núcleo tenha acesso às chamadas de sistema do sistema hospedeiro quando os seus próprios processos solicitarem este acesso, ao mesmo tempo em que impede os mesmos de acessar diretamente os recursos reais subjacentes.

No hipervisor, a distinção de privilégios foi implementada com o mecanismo de interceptação de chamadas do próprio Linux, fornecido pela chamada de sistema `ptrace`[3]. Usando a chamada `ptrace`, o hipervisor recebe o controle de todas as cha-

[3]Chamada de sistema que permite observar e controlar a execução de outros processos; o comando `strace` do Linux permite ter uma noção de como a chamada de sistema `ptrace` funciona.

madas de sistema de entrada/saída geradas pelas máquinas virtuais. Todos os sinais gerados ou enviados às máquinas virtuais também são interceptados. A chamada `ptrace` também é utilizada para manipular o contexto do sistema convidado.

O *User-Mode Linux* utiliza o sistema hospedeiro para operações de entrada/saída. Como a máquina virtual é um processo no sistema hospedeiro, a troca de contexto entre duas instâncias de máquinas virtuais é rápida, assim como a troca entre dois processos do sistema hospedeiro. Entretanto, modificações no sistema convidado foram necessárias para a otimização da troca de contexto. A virtualização das chamadas de sistema é implementada pelo uso de uma *thread* de rastreamento que intercepta e redireciona todas as chamadas de sistema para o núcleo virtual. Este identifica a chamada de sistema e os seus argumentos, cancela a chamada e modifica estas informações no hospedeiro, onde o processo troca de contexto e executa a chamada na pilha do núcleo.

O *User-Mode Linux* está disponível na versão 2.6 do núcleo Linux, ou seja, ele foi assimilado à árvore oficial de desenvolvimento do núcleo, portanto melhorias na sua arquitetura deverão surgir no futuro, ampliando seu uso em diversos contextos de aplicação.

4.3.5. QEMU

O *QEMU* é um hipervisor com virtualização completa [Bellard 2005]. Não requer alterações ou otimizações no sistema hospedeiro, pois utiliza intensivamente a tradução dinâmica (seção 4.1.5) como técnica para prover a virtualização. É um dos poucos hipervisores recursivos, ou seja, é possível chamar o *QEMU* a partir do próprio *QEMU*. O hipervisor *QEMU* oferece dois modos de operação:

- *Emulação total do sistema*: emula um sistema completo, incluindo processador (normalmente um *Intel Pentium II*) e vários periféricos. Neste modo o emulador pode ser utilizado para executar diferentes sistemas operacionais;
- *Emulação no modo de usuário*: disponível apenas para o sistema Linux. Neste modo o emulador pode executar processos Linux compilados em diferentes plataformas (por exemplo, um programa compilado para um processador *x86* pode ser executado em um processador *PowerPC* e vice-versa).

Durante a emulação de um sistema completo, o *QEMU* implementa uma MMU (*Memory Management Unit*) totalmente em software, para garantir o máximo de portabilidade. Quando em modo usuário, o *QEMU* simula uma MMU simplificada através da chamada de sistema `mmap` (que permite mapear um arquivo em uma região da memória) do sistema hospedeiro.

Por meio de um módulo instalado no núcleo do sistema hospedeiro, denominado *KQEMU* ou *QEMU Accelerator*, o hipervisor *QEMU* consegue obter um desempenho similar ao de outras máquinas virtuais como *VMWare* e *User-Mode Linux*. Com este módulo, o *QEMU* passa a executar as chamadas de sistema emitidas pelos processos convidados diretamente sobre o sistema hospedeiro, ao invés de interpretar cada uma. O *KQEMU* permite associar os dispositivos de entrada/saída e o endereçamento de memória

do sistema convidado aos do sistema hospedeiro. Processos em execução sobre o núcleo convidado passam a executar diretamente no modo usuário do sistema hospedeiro. O modo núcleo do sistema convidado é utilizado apenas para virtualizar o processador e os periféricos.

O *VirtualBox* [VirtualBox 2008] é um ambiente de máquinas virtuais construído sobre o hipervisor *QEMU*. Ele é similar ao *VMware Workstation* em muitos aspectos. Atualmente, pode tirar proveito do suporte à virtualização disponível nos processadores Intel e AMD. Originalmente desenvolvido pela empresa *Innotek*, o *VirtualBox* foi adquirido pela *Sun Microsystems* e liberado para uso público sob a licença *GPLv2*.

4.3.6. Valgrind

O *Valgrind* [Nethercote and Seward 2007] é uma ferramenta de depuração de uso da memória RAM e problemas correlatos. Ele permite investigar fugas de memória (*memory leaks*), acessos a endereços inválidos, padrões de uso dos caches e outras operações envolvendo o uso da memória RAM. O *Valgrind* foi desenvolvido para plataforma *x86 Linux*, mas existem versões experimentais para outras plataformas.

Tecnicamente, o *Valgrind* é um hipervisor de aplicação que virtualiza o processador através de técnicas de tradução dinâmica. Ao iniciar a análise de um programa, o *Valgrind* traduz o código executável do mesmo para um formato interno independente de plataforma denominado IR (*Intermediate Representation*). Após a conversão, o código em IR é instrumentado, através da inserção de instruções para registrar e verificar as operações de alocação, acesso e liberação de memória. A seguir, o programa IR devidamente instrumentado é traduzido no formato binário a ser executado sobre o processador virtual. O código final pode ser até 50 vezes mais lento que o código original, mas essa perda de desempenho normalmente não é muito relevante durante a análise de um programa.

4.3.7. JVM – Java Virtual Machine

É comum a implementação do suporte de execução de uma linguagem de programação usando uma máquina virtual. Um exemplo clássico nesse sentido é o compilador *UCSD Pascal*. Nesse sistema, um programa escrito em Pascal é compilado em um código binário independente de plataforma denominado *P-Code*, que executava sobre o processador abstrato *P-Machine*. O interpretador de *P-Codes* era bastante compacto e facilmente portável, o que tornou o sistema P muito popular nos anos 1970. A estrutura da *P-Machine* é orientada a *pilha*, ou seja, a maioria das instruções realiza suas operações usando a pilha ao invés de registradores específicos.

Um exemplo atual dessa abordagem ocorre na linguagem Java. Tendo sido originalmente concebida para o desenvolvimento de pequenos aplicativos e programas de controle de aparelhos eletroeletrônicos, a linguagem Java mostrou-se ideal para ser usada na Internet. O que o torna tão atraente é o fato de programas escritos em Java poderem ser executados em praticamente qualquer plataforma. A virtualização é o fator responsável pela independência dos programas Java do hardware e dos sistemas operacionais: um programa escrito em Java, ao ser compilado, gera um código binário específico para uma máquina abstrata denominada *máquina virtual Java* (*JVM - Java Virtual Machine*). A linguagem de máquina executada pela máquina virtual Java é denominada *bytecode*

Java, e não corresponde a instruções de nenhum processador real. A máquina virtual deve então interpretar todas as operações do *bytecode*, utilizando as instruções da máquina real subjacente para executá-las.

A vantagem mais significativa da abordagem adotada por Java é a *portabilidade* do código executável: para que uma aplicação Java possa executar sobre uma determinada plataforma, basta que a máquina virtual Java esteja disponível ali (na forma de um suporte de execução denominado JRE - *Java Runtime Environment*). Assim, a portabilidade dos programas Java depende unicamente da portabilidade da própria máquina virtual Java. O suporte de execução Java pode estar associado a um navegador Web, o que permite que código Java seja associado a páginas Web, na forma de pequenas aplicações denominadas *applets*, que são trazidas junto com os demais componentes de página Web e executam localmente no navegador. A figura 4.25 mostra os principais componentes da plataforma Java.

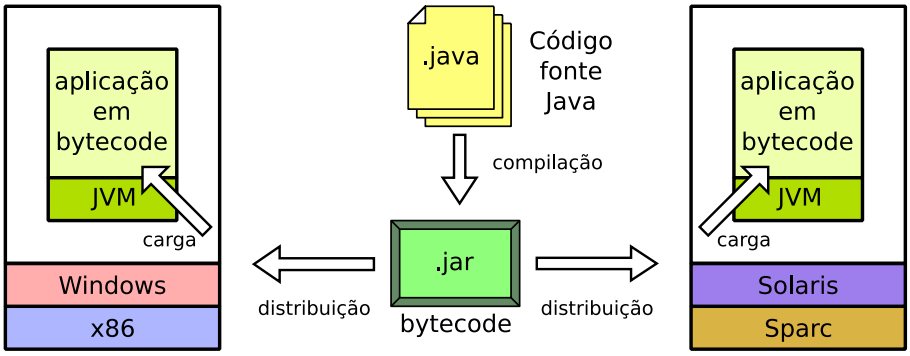

Figura 4.25. Máquina virtual Java.

É importante ressaltar que a adoção de uma máquina virtual como suporte de execução não é exclusividade de Java, nem foi inventada por seus criadores. As primeiras experiências de execução de aplicações sobre máquinas abstratas remontam aos anos 1970, com a linguagem *UCSD Pascal*. Hoje, muitas linguagens adotam estratégias similares, como Java, C#, Python, Perl, Lua e Ruby. Em C#, o código-fonte é compilado em um formato intermediário denominado CIL (*Common Intermediate Language*), que executa sobre uma máquina virtual CLR (*Common Language Runtime*). CIL e CLR fazem parte da infraestrutura .NET da Microsoft.

Em termos de desempenho, um programa compilado para uma máquina virtual executa mais lentamente que seu equivalente compilado sobre uma máquina real, devido ao custo de interpretação do *bytecode*. Todavia, essa abordagem oferece melhor desempenho que linguagens puramente interpretadas. Além disso, técnicas de otimização como a tradução dinâmica (ou compilação *Just-in-Time*), na qual blocos de instruções repetidos freqüentemente são compilados pelo monitor e armazenados em cache, permitem obter ganhos significativos de desempenho.

4.4. Perspectivas

A utilização de máquinas virtuais tornou-se uma alternativa concreta para várias soluções domésticas e corporativas. Graças a diversas pesquisas, no futuro será possível utilizar

os melhores recursos das mais variadas plataformas operacionais sem a necessidade de investir em equipamentos específicos. São vários os exemplos de sistemas que utilizam os conceitos de virtualização – alguns vistos neste trabalho -, mas, com certeza, vários outros exemplos surgirão no futuro para aproveitar os novos avanços nessa área.

Embora o uso de máquinas virtuais tenha evoluído, várias outras pesquisas na indústria e nas universidades devem ser realizadas para aprimorar as questões de segurança, mobilidade e desempenho dos hipervisores [Rosenblum and Garfinkel 2005]. Os principais campos de pesquisas nos próximos anos para melhorar o suporte à virtualização provavelmente serão:

- *Processadores*: os principais fabricantes de processadores, AMD e Intel, já disponibilizaram tecnologias para que a virtualização sobre a plataforma x86 ocorra de forma mais natural e tranqüila. Estas tecnologias vêm simplificando o desenvolvimento dos novos hipervisores.
- *Memória*: Várias técnicas têm permitido que a virtualização da memória seja mais eficiente. Pesquisas futuras devem levar aos sistemas operacionais convidados a gerenciar a memória juntamente com o hipervisor (gerência cooperativa).
- *Entrada/saída*: Os dispositivos de entrada/saída deverão ser projetados para fornecer suporte à virtualização com alto desempenho. O próprio dispositivo deverá suportar a multiplexação, de forma a permitir o acesso simultâneo por vários sistemas virtuais. A responsabilidade pelo acesso aos dispositivos deverá passar do hipervisor para o sistema convidado.

Várias pesquisas vêm sendo conduzidas sobre a aplicação de hipervisores na segurança de sistemas, mas também na melhoria da segurança dos próprios hipervisores. Alguns problemas que provavelmente merecerão a atenção dos pesquisadores nos próximos anos incluem:

- Garantir que as propriedades básicas dos hipervisores sejam providas corretamente pelas implementações; uma falha de segurança no hipervisor pode por em risco a segurança deste, do sistema hospedeiro e de todos os sistemas convidados;
- Construir métodos confiáveis para detectar a execução em sistemas convidados, para descobrir a presença de *rootkits* baseados em máquinas virtuais;
- De forma contraditória, criar técnicas para impedir a detecção de ambiente convidados, que serão úteis na construção de *honeypots* virtuais;
- Trazer para o *desktop* o conceito de *sandboxing*, que é a implementação de máquinas virtuais leves e transparentes ao usuário, para a execução isolada de aplicações sensíveis como navegadores Internet, clientes de e-mail e programas de mensagens instantâneas;

Agradecimentos

Os autores desejam agradecer as críticas e sugestões recebidas dos revisores da proposta deste minicurso; desejam também agradecer a Fundação Araucária – FAP do Estado do Paraná, pelo apoio financeiro recebido na forma de uma bolsa de doutorado.

Referências

[Barham et al. 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177.

[Bayer et al. 2006] Bayer, U., Moser, A., Kruegel, C., and Kirda, E. (2006). Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1).

[Beaucamps and Filiol 2007] Beaucamps, P. and Filiol, E. (2007). On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1):3–21.

[Bellard 2005] Bellard, F. (2005). QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*.

[Clark et al. 2005] Clark, C., Fraser, K., Hand, S., Hansen, J., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation*.

[Dike 2000] Dike, J. (2000). A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*, Atlanta - USA.

[Duesterwald 2005] Duesterwald, E. (2005). Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE*, 93(2):436–448.

[Dunlap et al. 2002] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. (2002). Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*.

[Ferre et al. 2006] Ferre, M. R., Pomeroy, D., Wahlstrom, M., and Watts, D. (2006). *Virtualization on the IBM System x3950 Server*. IBM RedBooks. http://www.redbooks.ibm.com.

[Fu and Xu 2005] Fu, S. and Xu, C.-Z. (2005). Service migration in distributed virtual machines for adaptive grid computing. In *34th IEEE Intl Conference on Parallel Processing*.

[Garfinkel et al. 2007] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. (2007). Compatibility is not transparency: VMM detecion myths and realities. Technical report, TRUST - The Team for Research in Ubiquitous Secure Technology.

[Garfinkel and Rosenblum 2003] Garfinkel, T. and Rosenblum, M. (2003). A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium*.

[Goldberg 1973] Goldberg, R. (1973). Architecture of virtual machines. *AFIPS National Computer Conference*.

[Goldberg and Mager 1979] Goldberg, R. and Mager, P. (1979). Virtual machine technology: A bridge from large mainframes to networks of small computers. *IEEE Proceedings Compcon Fall 79*, pages 210–213.

[Herder et al. 2006] Herder, J. N., Bos, H., Grass, B., Homburg, P., and Tanenbaum, A. S. (2006). Reorganizing UNIX for reliability. In *Proceedings of 11th Asia-Pacific Computer Systems Architecture Conference*, pages 81–94.

[Honeynet Project 2001] Honeynet Project (2001). *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley.

[Honeynet Project 2003] Honeynet Project (2003). Know your enemy: defining virtual honeynets. http://project.honeynet.org.

[IBM 2007] IBM (2007). *Power Instruction Set Architecture – Version 2.04*. IBM Corporation.

[Kamp and Watson 2000] Kamp, P.-H. and Watson, R. N. M. (2000). Jails: Confining the omnipotent root. In *Proceedings of the Second International System Administration and Networking Conference (SANE)*.

[King and Chen 2006] King, S. and Chen, P. (2006). Subvirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, pages 1–14.

[Klaus 1999] Klaus, B. (1999). From antivirus to antimalware software and beyond: Another approach to the protection of customers from dysfunctional system behaviour. In *22nd National Information Systems Security Conference*. http://csrc.nist.gov/nissc/1999/proceeding/papers/p12.pdf.

[Krause and Tipton 1999] Krause, M. and Tipton, H. F. (1999). *Handbook of Information Security Management*. Auerbach Publications.

[Kruegel et al. 2004] Kruegel, C., Robertson, W., and Vigna, G. (2004). Detecting kernel-level rootkits through binary analysis. In *20th Annual Computer Security Applications Conference*.

[Kwan and Durfee 2007] Kwan, P. C. S. and Durfee, G. (2007). Practical uses of virtual machines for protection of sensitive user data. In *Information Security Practice and Experience*, pages 145–161. Springer Berlin / Heidelberg.

[Laureano et al. 2004] Laureano, M., Maziero, C., and Jamhour, E. (2004). Intrusion detection in virtual machine environments. In *30th EUROMICRO Conference*, pages 520–525, Rennes - França.

[Laureano et al. 2007] Laureano, M., Maziero, C., and Jamhour, E. (2007). Protecting host-based intrusion detectors through virtual machines. *Computer Networks*, 51:1275–1283.

[Laureano 2006] Laureano, M. A. P. (2006). *Máquinas Virtuais e Emuladores - Conceitos, Técnicas e Aplicações*. Novatec Editora, first edition.

[LeVasseur et al. 2004] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. (2004). Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA.

[Liston and Skoudis 2006] Liston, T. and Skoudis, E. (2006). On the cutting edge: Thwarting virtual machine detection. In *SANS Conference*.

[Nanda and Chiueh 2005] Nanda, S. and Chiueh, T. (2005). A survey on virtualization technologies. Technical report, University of New York at Stony Brook.

[Nethercote and Seward 2007] Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego - California - USA.

[Newman et al. 2005] Newman, M., Wiberg, C.-M., and Braswell, B. (2005). *Server Consolidation with VMware ESX Server*. IBM RedBooks. http://www.redbooks.ibm.com.

[Ormandy 2007] Ormandy, T. (2007). An empirical study into the security exposure to hosts of hostile virtualized environments. In *CanSecWest*, Vancouver BC.

[Popek and Goldberg 1974] Popek, G. and Goldberg, R. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421.

[Provos 2004] Provos, N. (2004). A virtual honeypot framework. In *13th USENIX Security Symposium*, San Diego, CA.

[Raffetseder et al. 2007] Raffetseder, T., Kruegel, C., and Kirda, E. (2007). Detecting system emulators. In *10th Information Security Conference - LNCS*.

[Robin and Irvine 2000] Robin, J. and Irvine, C. (2000). Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*.

[Rosenblum 2004] Rosenblum, M. (2004). The reincarnation of virtual machines. *Queue Focus - ACM Press*, pages 34–40.

[Rosenblum and Garfinkel 2005] Rosenblum, M. and Garfinkel, T. (2005). Virtual machine monitors: current tecnology and future trends. *IEEE Computer Magazine*, 38(5):39–47.

[Rutkowska 2004] Rutkowska, J. (2004). Red pill... or how to detect VMM using (almost) one CPU instruction. http://invisiblethings.org/papers/redpill.html.

[Rutkowska 2006] Rutkowska, J. (2006). Subverting Vista kernel for fun and profit. http://www.blackhat.com. Black Hat USA 2006.

[Seward and Nethercote 2005] Seward, J. and Nethercote, N. (2005). Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*.

[Skoudis and Zeltser 2003] Skoudis, E. and Zeltser, L. (2003). *Malware: Fighting Malicious Code*. Prentice Hall PTR.

[Smith and Nair 2004] Smith, J. and Nair, R. (2004). *Virtual Machines: Architectures, Implementations and Applications*. Morgan Kaufmann.

[Smith and Nair 2005] Smith, J. E. and Nair, R. (2005). The architecture of virtual machines. *IEEE Computer*, pages 32–38.

[Sugerman et al. 2001] Sugerman, J., Venkitachalam, G., and Lim, B. H. (2001). Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 1–14.

[Sêmola 2003] Sêmola, M. (2003). *Gestão da Segurança da Informação - Uma visão executiva*. Campus, Rio de Janeiro.

[Tan et al. 2007] Tan, L., Chan, E. M., Farivar, R., Mallick, N., Carlyle, J. C., David, F. M., and Campbell, R. H. (2007). iKernel: Isolating buggy and malicious device drivers using hardware virtualization support. In *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 134–144.

[Uhlig et al. 2005] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A., Anderson, F. M. A., Bennett, S., Kägi, A., Leung, F., and Smith, L. (2005). Intel virtualization technology. *IEEE Computer*.

[Ung and Cifuentes 2006] Ung, D. and Cifuentes, C. (2006). Dynamic re-engineering of binary code with run-time feedbacks. *Science of Computer Programming*, 60(2):189–204.

[VirtualBox 2008] VirtualBox, I. (2008). The VirtualBox architecture. http://www.virtualbox.org/wiki/VirtualBox_architecture.

[VMware 2000] VMware (2000). VMware technical white paper. Technical report, VMware, Palo Alto, CA - USA.

[Whitaker et al. 2002] Whitaker, A., Shaw, M., and Gribble, S. (2002). A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint Emilion - France.

[Willems et al. 2007] Willems, C., Holz, T., and Freiling, F. (2007). Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39.

[Yen 2007] Yen, C.-H. (2007). Solaris operating system - hardware virtualization product architecture. Technical Report 820-3703-10, Sun Microsystems.

[Zovi 2006] Zovi, D. A. D. (2006). Hardware virtualization based rootkits. http://www.blackhat.com. Black Hat USA 2006.