Sistemas Operacionais: Conceitos e Mecanismos

IX - Virtualização

Prof. Carlos Alberto Maziero
DAInf UTFPR
http://dainf.ct.utfpr.edu.br/~maziero

2 de junho de 2013

Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto LaTeX 2$_\varepsilon$, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criadores de gráficos *GNUPlot* e *GraphViz* e processador PS/PDF *GhostScript*, entre outros.

Sumário

Resumo

As tecnologias de virtualização do ambiente de execução de aplicações ou de plataformas de hardware têm sido objeto da atenção crescente de pesquisadores, fabricantes de hardware/software, administradores de sistemas e usuários avançados. Os recentes avanços nessa área permitem usar máquinas virtuais com os mais diversos objetivos, como a segurança, a compatibilidade de aplicações legadas ou a consolidação de servidores. Este capítulo apresenta os principais conceitos, arquiteturas e implementações de ambientes virtuais de execução, como máquinas virtuais, emuladores e contêineres.

1 Conceitos básicos

As tecnologias de virtualização do ambiente de execução de aplicações ou de plataformas de hardware têm sido objeto da atenção crescente de pesquisadores, fabricantes de hardware/software, administradores de sistemas e usuários avançados. A virtualização de recursos é um conceito relativamente antigo, mas os recentes avanços nessa área permitem usar máquinas virtuais com os mais diversos objetivos, como a segurança, a compatibilidade de aplicações legadas ou a consolidação de servidores. Este capítulo apresenta os principais conceitos, arquiteturas e técnicas usadas para a implementação de ambientes virtuais de execução.

1.1 Um breve histórico

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 1970, os pesquisadores Popek & Goldberg formalizaram vários conceitos associados às máquinas virtuais, e definiram as condições necessárias para que uma plataforma de hardware suporte de forma eficiente a virtualização [Popek and Goldberg, 1974]; essas condições são discutidas em detalhe na Seção 2.1. Nessa mesma época surgem as primeiras experiências concretas de utilização de máquinas virtuais para a execução de aplicações, com o ambiente *UCSD p-System*, no qual programas Pascal são compilados para execução sobre um hardware abstrato denominado *P-Machine*.

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.

1.2 Interfaces de sistema

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. 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. As interfaces existentes entre os componentes de um sistema de computação típico 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 1 apresenta essa visão conceitual da arquitetura de um sistema computacional, com seus vários componentes e as respectivas interfaces entre eles.

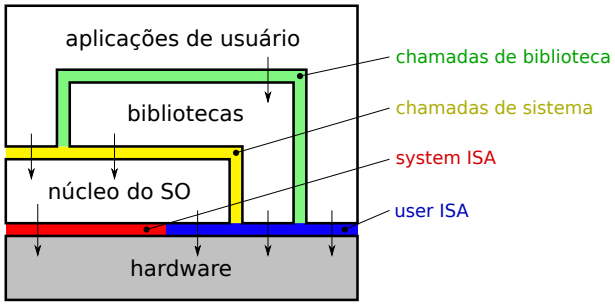

Figura 1: Componentes e interfaces de um sistema computacional.

1.3 Compatibilidade entre interfaces

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 2 representa essas duas interfaces.

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

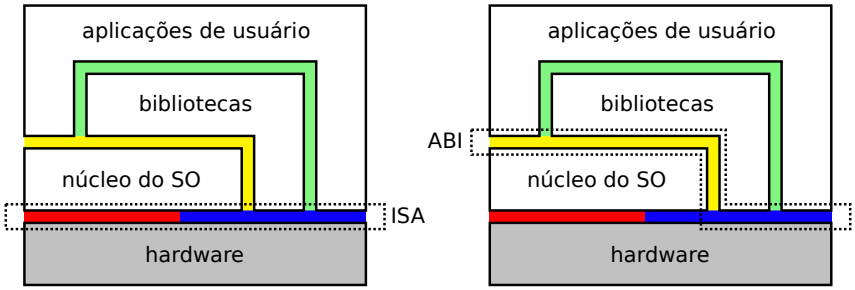

Figura 2: Interfaces de sistema ISA e ABI [Smith and Nair, 2004].

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 3, extraída de [Smith and Nair, 2004], ilustra esses problemas de compatibilidade entre interfaces.

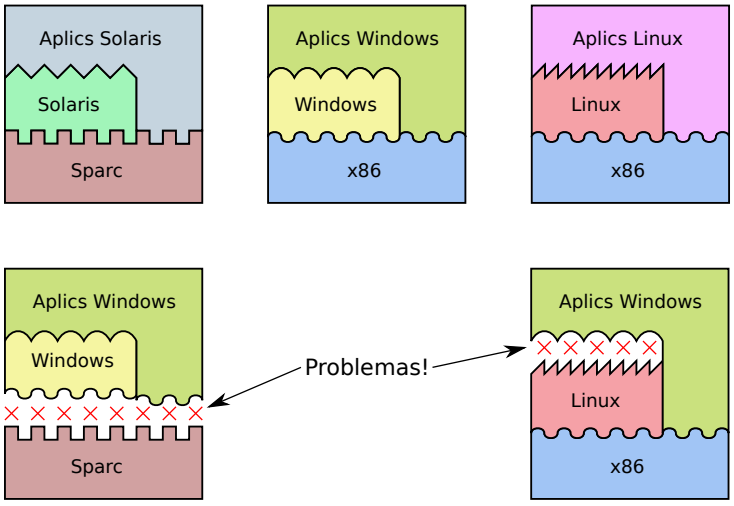

Figura 3: Problemas de compatibilidade entre interfaces [Smith and Nair, 2004].

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 consequência). 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 da virtualização, que será apresentada na próxima seção.

1.4 Virtualização de interfaces

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 um sistema Windows, pois as chamadas de sistema emitidas pelo programa Windows não serão compreendidas pelo sistema operacional Linux 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.

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, 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).

Um ambiente de máquina virtual consiste de três partes básicas, que podem ser observadas na Figura 4:

- O sistema real, nativo ou hospedeiro (*host system*), que contém os recursos reais de hardware e software do sistema;

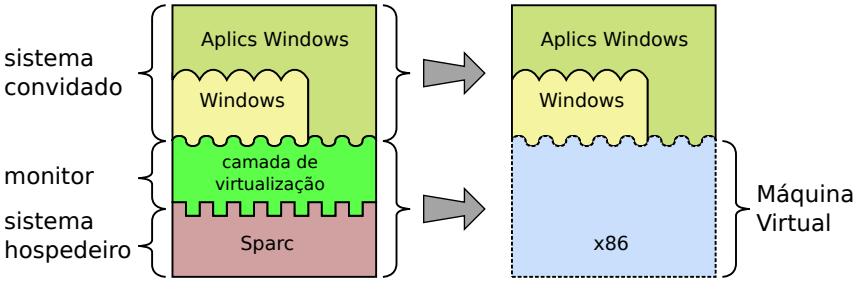

Figura 4: Uma máquina virtual [Smith and Nair, 2004].

- 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, chamada *hipervisor* ou *monitor* (VMM – *Virtual Machine Monitor*), que constrói as interfaces virtuais a partir da interface real.

É importante ressaltar a diferença entre os termos *virtualização* e *emulação*. A emulação é na verdade uma forma de virtualização: quando um hipervisor virtualiza integralmente uma interface de hardware ou de sistema operacional, é geralmente chamado de *emulador*. Por exemplo, a máquina virtual Java, que constrói um ambiente completo para a execução de *bytecodes* a partir de um processador real que não executa *bytecodes*, pode ser considerada um emulador.

A virtualização abre uma série de possibilidades interessantes para a composição de um sistema de computação, como por exemplo (Figura 5):

- *Emulação de hardware*: um sistema operacional convidado e suas aplicações, desenvolvidas para uma plataforma de hardware A, são executadas sobre uma plataforma de hardware distinta B.
- *Emulação de sistema operacional*: aplicações construídas para um sistema operacional X são executadas sobre outro sistema operacional Y.
- *Otimização 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.

1.5 Virtualização versus abstração

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 operacionais [Smith and Nair, 2004]. Um dos principais objetivos dos sistemas operacionais é oferecer uma visão de alto nível dos recursos de hardware, que seja mais

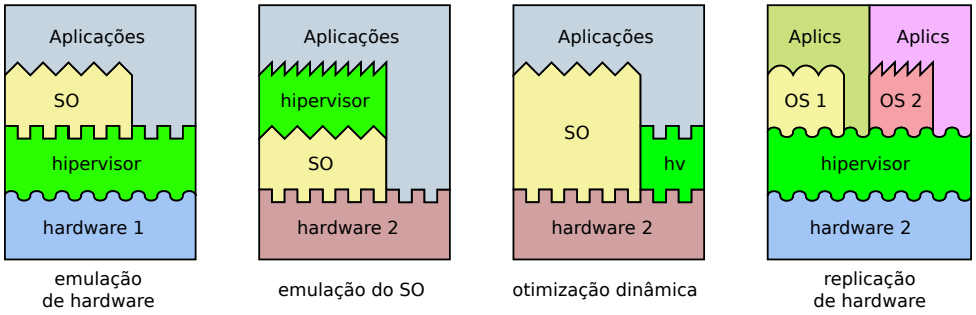

Figura 5: Possibilidades de virtualização [Smith and Nair, 2004].

simples de usar e menos dependente das tecnologias subjacentes. Essa visão abstrata dos recursos é construída de forma incremental, em níveis de abstração crescentes. Exemplos típicos dessa estruturação em níveis de abstração 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 6 apresenta os níveis de abstração de um subsistema de gerência de disco típico.

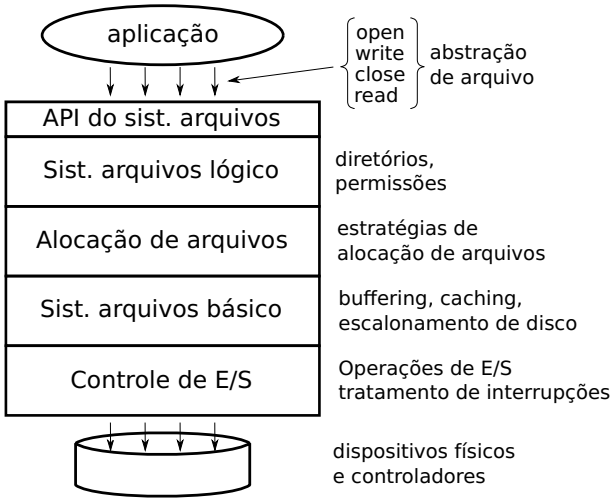

Figura 6: Níveis de abstração em um sub-sistema de disco.

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 7 ilustra essa diferença: através da virtualização, um processador Sparc pode ser visto pelo sistema convidado 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).

A Figura 8 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

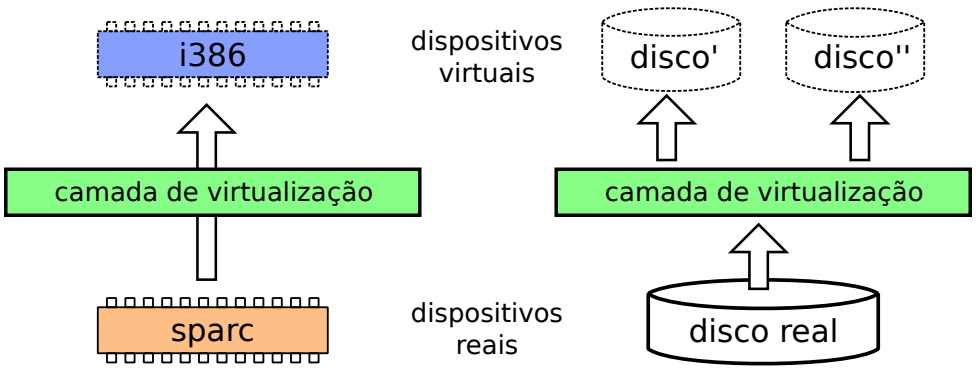

Figura 7: Virtualização de recursos do hardware.

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.

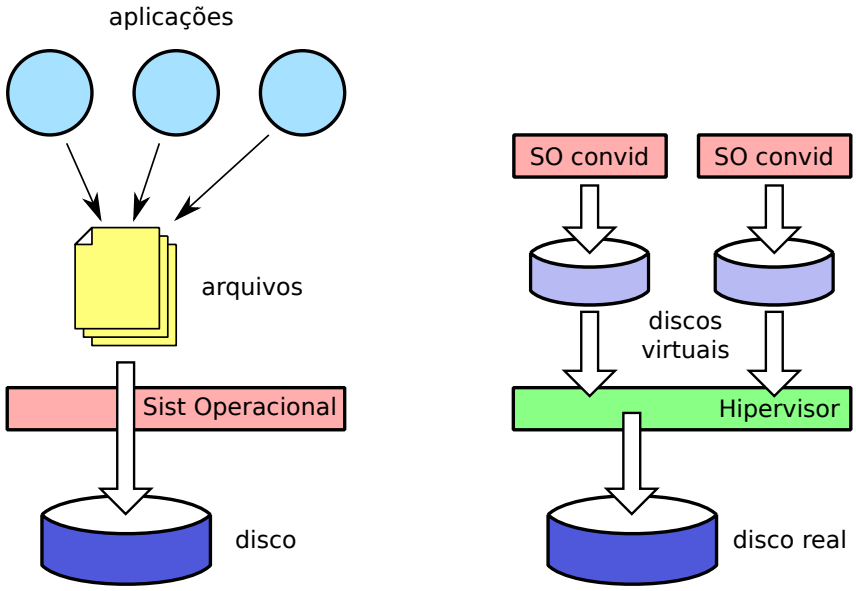

Figura 8: Abstração versus virtualização de um disco rígido.

2 A construção de máquinas virtuais

Conforme apresentado, a virtualização consiste em reescrever uma ou mais interfaces do sistema computacional, para oferecer novas interfaces e assim permitir a execução de sistemas operacionais ou aplicações incompatíveis com as interfaces originais.

A construção de máquinas virtuais é bem mais complexa que possa parecer à primeira vista. Caso os conjuntos de instruções (ISA) 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 fim, 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.

Esta seção aborda inicialmente o conceito formal de virtualização, para em seguida discutir as principais técnicas usadas na construção de máquinas virtuais.

2.1 Definição formal

Em 1974, os pesquisadores americanos Gerald Popek (UCLA) e Robert Goldberg (Harvard) definiram uma máquina virtual da seguinte forma [Popek and Goldberg, 1974]:

> *Uma máquina virtual é vista como uma duplicata eficiente e isolada de uma máquina real. Essa abstração é construída por um "monitor de máquina virtual" (VMM - Virtual Machine Monitor).*

O hipervisor ou monitor de máquina virtual descrito por Popek/Goldberg corresponde à camada de virtualização apresentada na Seção 1.4. Para funcionar de forma correta e eficiente, o 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 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 retirar recursos previamente alocados à máquina virtual.

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 processador real.

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

Isolamento: aplicações dentro de uma máquina virtual não podem interagir diretamente (a) com outras máquinas virtuais, (b) com o hipervisor, ou (c) com o sistema real hospedeiro. Todas as interações entre entidades dentro de uma máquina virtual e o mundo exterior devem ser mediadas pelo hipervisor.

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*.

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.

Essas 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 do processador: 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) deve ser 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 as restrições de Popek/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 presentes nos programas de usuário: ao carregar um programa na memória, o hipervisor analisa seu código e substitui essas instruções sensíveis 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*, que se baseia em reescrever parte do sistema convidado para não usar essas instruções sensíveis. Ambas as técnicas são discutidas a seguir.

2.2 Suporte de hardware

Na época em que Popek e Goldberg definiram seu principal teorema, o hardware 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, como os mainframes, 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 (Seção 2.1) e dificulta a criação de máquinas virtuais em sistemas que usam esses processadores [Robin and Irvine, 2000]. Alguns exemplos dessas instruções "problemáticas" são:

- `SGDT/SLDT`: permitem ler o registrador que indica a posição e tamanho das tabelas de segmentos global/local do processo ativo.
- `SMSW`: permite ler o registrador de controle 0, que contém informações de status interno do processador.
- `PUSHF/POPF`: empilha/desempilha o valor do registrador `EFLAGS`, que também contém informações de status interno do processador.

Para controlar o acesso aos recursos do sistema e às instruções privilegiadas, os processadores atuais usam a noção de "anéis de proteção" herdada do sistema MULTICS [Corbató and Vyssotsky, 1965]. Os anéis definem níveis de privilégio: um código executando no nível 0 (anel central) tem acesso completo ao hardware, enquanto um

código executando em um nível $i > 0$ (anéis externos) tem menos privilégio. Quanto mais externo o anel onde um código executa, menor o seu nível de privilégio. Os processadores Intel/AMD atuais suportam 4 anéis ou níveis de proteção, mas a quase totalidade dos sistemas operacionais de mercado somente usa os dois anéis extremos: o anel 0 para o núcleo do sistema e o anel 3 para as as aplicações dos usuários.

As técnicas de virtualização para as plataformas Intel/AMD se baseiam na redução de privilégios do sistema operacional convidado: o hipervisor e o sistema operacional hospedeiro executam no nível 0, o sistema operacional convidado executa no nível 1 ou 2 e as aplicações do sistema convidado executam no nível 3. Essas formas de estruturação de sistema são denominadas "modelo 0/1/3" e modelo "0/2/3", respectivamente (Figura 9). Todavia, para que a estratégia de redução de privilégio possa funcionar, algumas instruções do sistema operacional convidado devem ser reescritas dinamicamente, em tempo de carga do sistema convidado na memória, pois ele foi construído para executar no nível 0.

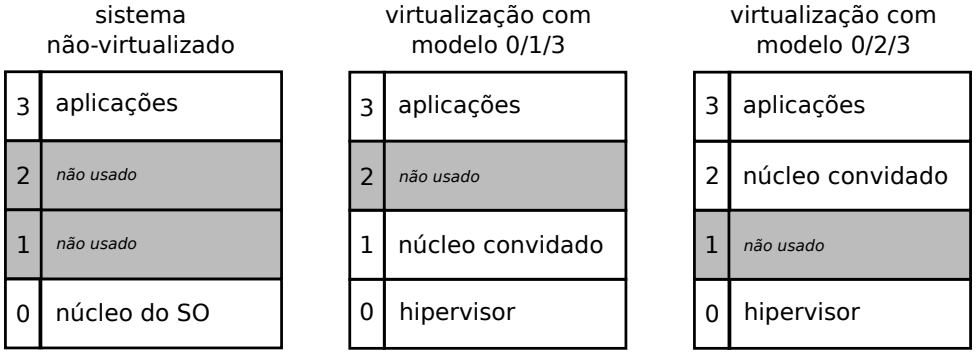

Figura 9: Uso dos níveis de proteção em processadores Intel/AMD convencionais.

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. Ambos os modos suportam os quatro níveis de privilégio, o que permite executar os sistemas convidados sem a necessidade de reescrita dinâmica de seu código.

São também definidos dois procedimentos de transição entre modos: *VM entry* (transição *root* → *non-root*) e *VM exit* (transição *non-root* → *root*). Quando operando dentro de uma máquina virtual (ou seja, em modo *non-root*), as instruções sensíveis e as interrupções podem provocar a transição *VM exit*, devolvendo o processador ao hipervisor em modo *root*. As instruções e interrupções que provocam a transição *VM exit* são configuráveis pelo próprio hipervisor.

Para gerenciar o estado do processador (conteúdo dos registradores), é definida uma *Estrutura de Controle de Máquina Virtual* (VMCS - *Virtual-Machine Control Structure*). Essa estrutura de dados contém duas áreas: uma para os sistemas convidados e outra para

o hipervisor. Na transição *VM entry*, o estado do processador é lido a partir da área de sistemas convidados da VMCS. Já uma transição *VM exit* faz com que o estado do processador seja salvo na área de sistemas convidados e o estado anterior do hipervisor, previamente salvo na VMCS, seja restaurado. A Figura 10 traz uma visão geral da arquitetura Intel IVT.

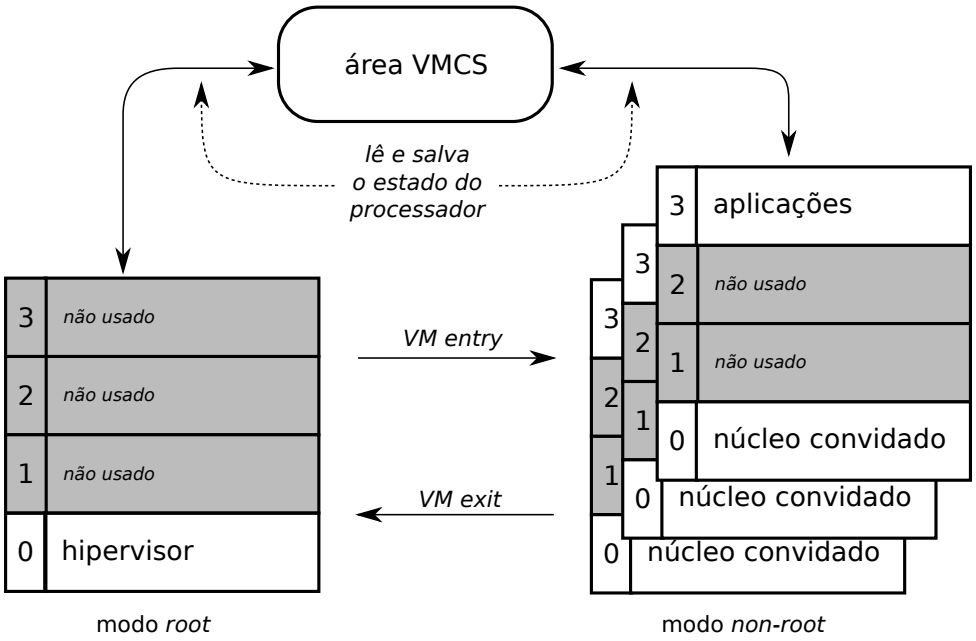

Figura 10: Visão geral da arquitetura *Intel IVT*.

Além da 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.

Conforme apresentado, a virtualização do processador pode obtida por reescrita dinâmica do código executável ou através do suporte nativo em hardware, usando tecnologias como IVT e AMD-V. Por outro lado, a virtualização da memória envolve outros desafios, que exigem modificações significativas dos mecanismos de gestão de memória virtual estudados na Seção **??**. Por exemplo, é importante prever o compartilhamento de páginas de código entre máquinas virtuais, para reduzir a quantidade total de memória física necessária a cada máquina virtual. Outros desafios similares surgem na virtualização dos dispositivos de armazenamento e de entrada/saída, alguns deles sendo analisados em [Rosenblum and Garfinkel, 2005].

2.3 Formas de virtualização

A virtualização implica na reescrita de interfaces de sistema, para permitir a interação entre componentes de sistema construídos para plataformas distintas. Como existem

várias interfaces entre os componentes, também há várias possibilidades de uso da virtualização em um sistema. De acordo com as interfaces em que são aplicadas, as formas mais usuais de virtualização são [Rosenblum, 2004, Nanda and Chiueh, 2005]:

Virtualização do hardware : toda a interface ISA, que permite o acesso ao hardware, é virtualizada. Isto inclui o conjunto de instruções do processador e as interfaces de acesso aos dispositivos de entrada/saída. A virtualização de hardware (ou virtualização completa) permite executar um sistema operacional e/ou aplicações em uma plataforma totalmente diversa daquela para a qual estes foram desenvolvidos. Esta é a forma de virtualização mais poderosa e flexível, mas também a de menor desempenho, uma vez que o hipervisor tem de traduzir toda as instruções geradas no sistema convidado em instruções do processador real. A máquina virtual Java (JVM, Seção 6.7) é um bom exemplo de virtualização do hardware. Máquinas virtuais implementando esta forma de virtualização são geralmente denominados *emuladores de hardware*.

Virtualização da interface de sistema : virtualiza-se a *System ISA*, que corresponde ao conjunto de instruções sensíveis do processador. Esta forma de virtualização é bem mais eficiente que a anterior, pois o hipervisor apenas emula as instruções sensíveis do processador virtual, executadas em modo privilegiado pelo sistema operacional convidado. As instruções não-sensíveis podem ser executadas diretamente pelo processador real, sem perda de desempenho. Todavia, apenas sistemas convidados desenvolvidos para o mesmo processador podem ser executados usando esta abordagem. Esta é a abordagem clássica de virtualização, presente nos sistemas de grande porte (*mainframes*) e usada nos ambientes de máquinas virtuais VMWare, VirtualPC e Xen.

Virtualização de dispositivos de entrada/saída : virtualizam-se as dispositivos físicos que permitem ao sistema interagir com o mundo exterior. Esta técnica implica na construção de dispositivos físicos virtuais, como discos, interfaces de rede e terminais de interação com o usuário, usando os dispositivos físicos subjacentes. A maioria dos ambientes de máquinas virtuais usa a virtualização de dispositivos para oferecer discos rígidos e interfaces de rede virtuais aos sistemas convidados.

Virtualização do sistema operacional : virtualiza-se o conjunto de recursos lógicos oferecidos pelo sistema operacional, como árvores de diretórios, descritores de arquivos, semáforos, canais de IPC e nomes de usuários e grupos. Nesta abordagem, cada máquina virtual pode ser vista como uma instância distinta do mesmo sistema operacional subjacente. Esta é a abordagem comumente conhecida como *servidores virtuais*, da qual são bons exemplos os ambientes *FreeBSD Jails*, *Linux VServers* e *Solaris Zones*.

Virtualização de chamadas de sistema : permite oferecer o conjunto de chamadas de sistema de uma sistema operacional A usando as chamadas de sistema de um sistema operacional B, permitindo assim a execução de aplicações desenvolvidas para um sistema operacional sobre outro sistema. Todavia, como as chamadas de sistema são normalmente invocadas através de funções de bibliotecas, a

virtualização de chamadas de sistema pode ser vista como um caso especial de virtualização de bibliotecas.

Virtualização de chamadas de biblioteca : tem objetivos similares ao da virtualização de chamadas de sistema, permitindo executar aplicações em diferentes sistemas operacionais e/ou com bibliotecas diversas daquelas para as quais foram construídas. O sistema *Wine*, que permite executar aplicações Windows sobre sistemas UNIX, usa essencialmente esta abordagem.

Na prática, essas várias formas de virtualização podem ser usadas para a resolução de problemas específicos em diversas áreas de um sistema de computação. Por exemplo, vários sistemas operacionais oferecem facilidades de virtualização de dispositivos físicos, como por exemplo: interfaces de rede virtuais que permitem associar mais de um endereço de rede ao computador, discos rígidos virtuais criados em áreas livres da memória RAM (os chamados *RAM disks*); árvores de diretórios virtuais criadas para confinar processos críticos para a segurança do sistema (através da chamada de sistem `chroot`), emulação de operações 3D em uma placa gráfica que não as suporta, etc.

3 Tipos de máquinas virtuais

Além de resolver problemas em áreas específicas do sistema operacional, as várias formas de virtualização disponíveis podem ser combinadas para a construção de *máquinas virtuais*. Uma máquina virtual é um ambiente de suporte à execução de software, construído usando uma ou mais formas de virtualização. Conforme as características do ambiente virtual proporcionado, as máquinas virtuais podem ser classificadas em três categorias, representadas na Figura 11:

Máquinas virtuais de processo (*Process Virtual Machines*): também chamadas de máquinas virtuais de aplicação, são ambientes construídos para prover suporte de execução a apenas um processo ou aplicação convidada específica. A máquina virtual Java e o ambiente de depuração *Valgrind* são exemplos deste tipo de ambiente.

Máquinas virtuais de sistema operacional (*Operating System Virtual Machines*): são construídas para suportar espaços de usuário distintos sobre um mesmo sistema operacional. Embora compartilhem o mesmo núcleo, cada ambiente virtual possui seus próprios recursos lógicos, como espaço de armazenamento, mecanismos de IPC e interfaces de rede distintas. Os sistemas *Solaris Zones* e *FreeBSD Jails* implementam este conceito.

Máquinas virtuais de sistema (*System Virtual Machines*): são ambientes de máquinas virtuais construídos para emular uma plataforma de hardware completa, com processador e periféricos. Este tipo de máquina virtual suporta sistemas operacionais convidados com aplicações convidadas executando sobre eles. Como exemplos desta categoria de máquinas virtuais temos os ambientes *VMware* e *VirtualBox*.

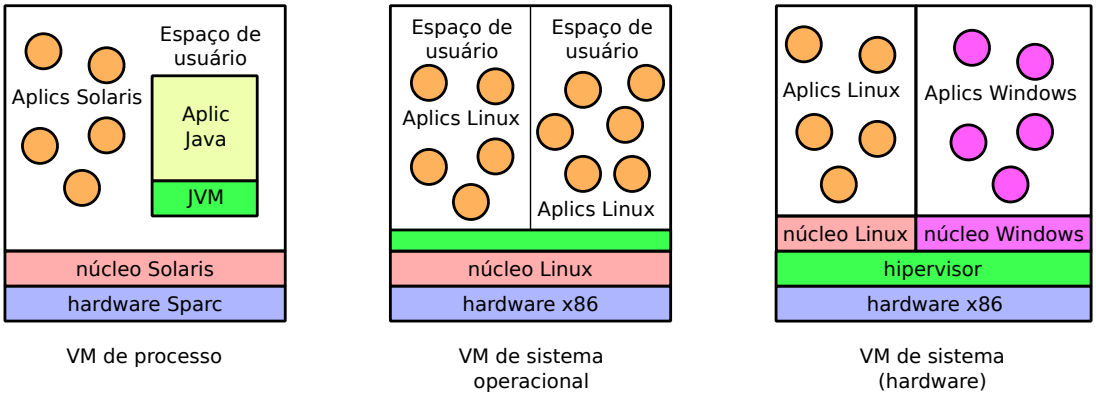

Figura 11: Máquinas virtuais de processo, de sistema operacional e de hardware.

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 - *Instruction Set Architecture*, Seção 1.2):

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 nenhuma relação com a interface de hardware do sistema real, ou seja, implementa um conjunto de instruções distinto, que deve ser totalmente traduzido pelo hipervisor. Conforme visto na Seção 2.1, a interpretação de instruções impõe um custo de execução significativo ao sistema convidado. A máquina virtual Java e o ambiente *QEmu* são exemplos dessa abordagem.

3.1 Máquinas virtuais de processo

Uma máquina virtual de processo ou de aplicação (*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 normalmente visto como um único processo dentro do sistema operacional subjacente (ou um pequeno conjunto de processos), submetido às mesmas condições e restrições que os demais processos nativos.

Os hipervisores que implementam máquinas virtuais de processo 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. Estas características violam a

propriedade de *isolamento* descrita na Seção 2.1, mas são necessárias para que a aplicação convidada se comporte como uma aplicação normal aos olhos do usuário.

Ao criar a máquina virtual para uma aplicação, o hipervisor pode implementar a mesma interface de hardware (ISA, Seção 1.2) 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, estudados no Capítulo **??**, também podem ser vistos como ambientes de máquinas virtuais. Em um sistema multi-tarefas, cada processo recebe um *processador virtual* (simulado através das fatias de tempo do processador real e das trocas de contexto), uma *memória virtual* (através do espaço de endereços mapeado para aquele processo) e *recursos físicos* (acessíveis através de chamadas de sistema). Este ambiente de virtualização é tão antigo e tão presente em nosso cotidiano que costumamos ignorá-lo como tal. 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 binário 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.

Contudo, as máquinas virtuais de processo mais populares atualmente são aquelas em que a interface binária de aplicação (ABI, Seção 1.2) requerida pela aplicação é diferente daquela 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 esses componentes. Nos dois casos, o hipervisor terá de fazer 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 e de bibliotecas usadas pela aplicação sobre as chamadas equivalentes 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 12).

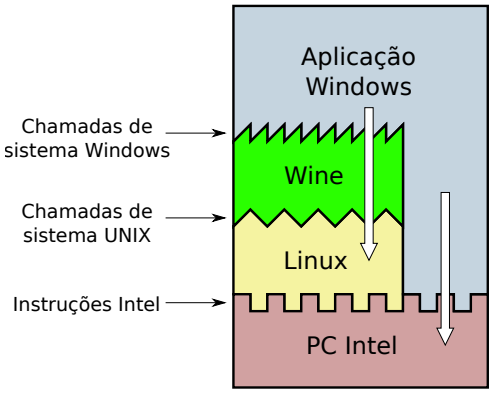

Figura 12: Funcionamento do emulador Wine.

Entretanto, muitas vezes a interface ISA utilizada pela aplicação não corresponde a nenhum hardware existente, mas a 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 um processador real. A máquina virtual deve então interpretar todas as operações do *bytecode*, utilizando as instruções da máquina real subjacente para executá-las. Várias linguagens empregam a mesma abordagem (embora usando um *bytecode* próprio), como Perl, Python e Smalltalk.

Em termos de desempenho, um programa compilado para um processador abstrato executa mais lentamente que seu equivalente compilado para um processador 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 frequentemente são traduzidos e mantidos em cache pelo hipervisor, permitem obter ganhos de desempenho significativos.

3.2 Máquinas virtuais de sistema operacional

Em muitas situações, a principal ou mesmo única motivação para o uso de máquinas virtuais é a propriedade de isolamento (Seção 2.1). Esta propriedade tem uma grande

importância no contexto da segurança de sistemas, por permitir isolar entre si sub-sistemas independentes que executam sobre o mesmo hardware. Por exemplo, a estratégia organizacional conhecida como *consolidação de servidores* advoga o uso de máquinas virtuais para abrigar os diversos servidores (de nomes, de arquivos, de e-mail, de Web) de um determinado domínio. Dessa forma, pode-se fazer um uso mais eficiente do hardware disponível, preservando o isolamento entre os serviços. Todavia, o impacto da virtualização de uma plataforma de hardware ou sistema operacional sobre o desempenho do sistema final pode ser elevado. As principais fontes desse impacto são a virtualização dos recursos (periféricos) da máquina real e a necessidade de tradução binária dinâmica das instruções do processador real.

Uma forma simples e eficiente de implementar o isolamento entre aplicações ou sub-sistemas em um sistema operacional consiste na virtualização do espaço de usuário (*userspace*). Nesta abordagem, denominada *máquinas virtuais de sistema operacional* ou *servidores virtuais*, o espaço de usuário do sistema operacional é dividido em áreas isoladas denominadas *domínios* ou *zonas* virtuais. A cada domínio virtual é alocada uma parcela dos recursos do sistema operacional, como memória, tempo de processador e espaço em disco. Além disso, alguns recursos do sistema real podem ser virtualizados, como é o caso frequente das interfaces de rede: cada domínio tem sua própria interface virtual e, portanto, seu próprio endereço de rede. Em várias implementações, cada domínio virtual define seu próprio espaço de nomes: assim, é possível encontrar um usuário `pedro` no domínio d_3 e outro usuário `pedro` no domínio d_7, sem conflitos. Essa noção de espaços de nomes distintos pode se estender aos demais recursos do sistema: identificadores de processos, semáforos, árvores de diretórios, etc.

Os processos presentes em um determinado domínio virtual 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 presentes em um domínio não podem ver ou interagir com processos que estiverem em outro domínio, não podem mudar de domínio, criar processos em outros domínios, nem consultar ou usar recursos de outros domínios. Dessa forma, para um determinado domínio, os demais domínios são máquinas distintas, acessíveis somente através de seus endereços de rede. Para fins de gerência, normalmente é definido um domínio d_0, chamado de *domínio inicial*, *privilegiado* ou *de gerência*, cujos processos têm visibilidade e acesso aos recursos dos demais domínios. Somente processos no domínio d_0 podem migrar para outros domínios; uma vez realizada uma migração, não há possibilidade de retornar ao domínio anterior.

O núcleo do sistema operacional é o mesmo para todos os domínios virtuais, e sua interface (conjunto de chamadas de sistema) é preservada. Normalmente apenas uma nova chamada de sistema é necessária, para que um processo no domínio inicial d_0 possa solicitar sua migração para um outro domínio d_i. A Figura 13 mostra a estrutura típica de um ambiente de máquinas virtuais de sistema operacional. Nela, pode-se observar que um processo pode migrar de d_0 para d_1, mas que os 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.

Há várias implementações disponíveis de mecanismos para a criação de domínios virtuais. A técnica mais antiga é implementada pela chamada de sistema `chroot`,

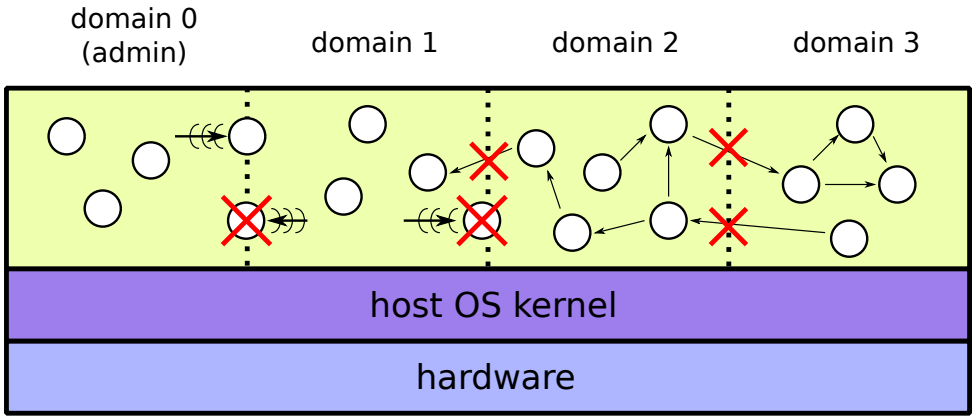

Figura 13: Máquinas virtuais de sistema operacional.

disponível na maioria dos sistemas UNIX. Essa chamada de sistema atua exclusivamente sobre o acesso de um processo ao sistema de arquivos: o processo que a executa tem seu acesso ao sistema de arquivos restrito a uma sub-árvore da hierarquia de diretórios, ou seja, ele fica "confinado" a essa sub-árvore. Os filhos desse processo herdam a mesma visão do sistema de arquivos, que não pode ser revertida. Por exemplo, um processo que executa a chamada `chroot ("/var/spool/postfix")` passa a ver somente a hierarquia de diretórios a partir do diretório `/var/spool/postfix`, que passa a ser seu diretório raiz. A Figura 14 ilustra essa operação.

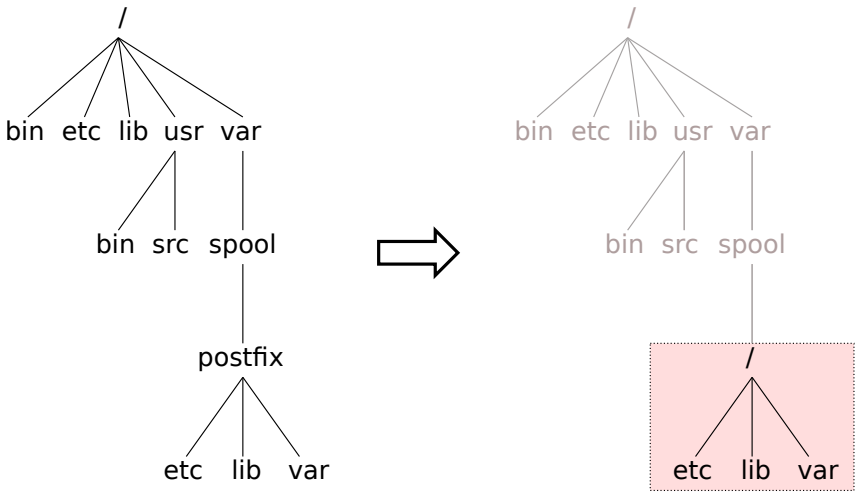

Figura 14: A chamada de sistema `chroot ("/var/spool/postfix")`.

A chamada de sistema `chroot` é muito utilizada para isolar processos que oferecem serviços à rede, como servidores DNS e de e-mail. Se um processo servidor tiver alguma vulnerabilidade e for subvertido por um atacante, este só terá acesso ao conjunto de diretórios visíveis aos processos do servidor, mantendo fora de alcance o restante da árvore de diretórios do sistema.

O sistema operacional *FreeBSD* oferece uma implementação de domínios virtuais mais elaborada, conhecida como *Jails* [McKusick and Neville-Neil, 2005] (aqui traduzidas

como *celas*). 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. Os processos dentro de uma cela estão submetidos às seguintes restrições:

- a visão do sistema operacional se restringe aos processos e recursos associados àquela cela; os demais processos, arquivos e outros recursos do sistema não-associados à cela não são visíveis;
- somente são permitidas interações (comunicação e coordenação) entre processos dentro da mesma cela;
- de forma similar à chamada `chroot`, cada cela recebe uma árvore de diretórios própria; operações de montagem/desmontagem de sistemas de arquivos são proibidas;
- cada cela tem um endereço de rede associado, que é o único utilizável pelos processos da cela; a configuração de rede (endereço, parâmetros de interface, tabela de roteamento) não pode ser modificada;
- não podem ser feitas alterações no núcleo do sistema, como reconfigurações ou inclusões/exclusões de módulos.

Essas restrições são impostas a todos os processos dentro de uma cela, mesmo aqueles pertencentes ao administrador (usuário *root*). Assim, uma cela constitui uma unidade de isolamento bastante robusta, que pode ser usada para confinar serviços de rede e aplicações ou usuários considerados "perigosos".

Existem implementações de estruturas similares às celas do *FreeBSD* em outros sistemas operacionais. Por exemplo, o sistema operacional *Solaris* implementa o conceito de *zonas* [Price and Tucker, 2004], que oferecem uma capacidade de isolamento similar à celas, além de prover um mecanismo de controle da distribuição dos recursos entre as diferentes zonas existentes. Outras implementações podem ser encontradas para o sistema Linux, como os ambientes *Virtuozzo/OpenVZ* e *Vservers/FreeVPS*.

3.3 Máquinas virtuais de sistema

Uma máquina virtual de sistema (ou de hardware) provê uma interface de hardware completa para 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 exclusiva. 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.

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 computadores separados. 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.

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 (conforme apresentado na Seção 2.1). 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 robustos 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 basicamente duas arquiteturas de hipervisores de sistema, apresentados na Figura 15:

Hipervisores nativos (ou *de tipo I*): nesta categoria, o hipervisor executa diretamente sobre o hardware do computador real, sem um sistema operacional subjacente. A função do hipervisor é virtualizar os recursos do 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 um computador completo 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, como processos separados. 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 operacionais previamente instalados, e podem ser facilmente lançados sob demanda. Por outro lado, hipervisores convidados têm desempenho pior que hipervisores nativos, pois têm de usar os recursos oferecidos pelo sistema operacional subjacente, enquanto

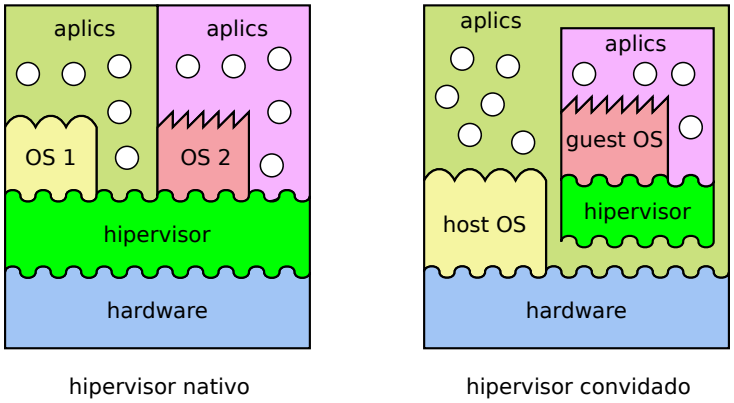

Figura 15: Arquiteturas de máquinas virtuais de sistema.

um hipervisor nativo pode acessar diretamente o hardware real. Técnicas para atenuar este problema são discutidas na Seção 4.5.

4 Técnicas de virtualização

A construção de hipervisores implica na definição de algumas estratégias para a virtualização. As estratégias mais utilizadas atualmente são a emulação completa do hardware, a virtualização da interface de sistema, a tradução dinâmica de código e a para-virtualização. Além disso, algumas técnicas complementares são usadas para melhorar o desempenho dos sistemas de máquinas virtuais. Essas técnicas são discutidas nesta seção.

4.1 Emulação completa

Nesta abordagem, toda a interface do hardware é virtualizada, incluindo todas as instruções do processador, a memória e os dispositivos periféricos. Isso permite oferecer ao sistema operacional convidado uma interface de hardware distinta daquela fornecida pela máquina real subjacente, caso seja necessário. O custo de virtualização pode ser muito elevado, pois cada instrução executada pelo sistema convidado tem de ser analisada e traduzida em uma ou mais instruções equivalentes no computador real. No entanto, esta abordagem permite executar sistemas operacionais em outras plataformas, distintas daquela para a qual foram projetados, sem nenhuma modificação.

Exemplos típicos de emulação completa 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.

Um caso especial de emulação completa consiste nos **hipervisores embutidos no hardware** (*codesigned hypervisors*). Um hipervisor embutido é visto como parte integrante do hardware e implementa a interface de sistema (ISA) vista pelos sistemas operacionais e aplicações daquela plataforma. Entretanto, 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 (vide Seção 4.3) para tratar as instruções executadas pelos 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 16).

4.2 Virtualização da interface de sistema

Nesta abordagem, a interface ISA de usuário é mantida, apenas as instruções privilegiadas e os dispositivos (discos, interfaces de rede, etc.) são virtualizados. Dessa forma, o sistema operacional convidado e as aplicações convidadas vêem o processador real. Como a quantidade de instruções a virtualizar é reduzida, o desempenho do sistema convidado pode ficar próximo daquele obtido se ele estivesse executando diretamente sobre o hardware real.

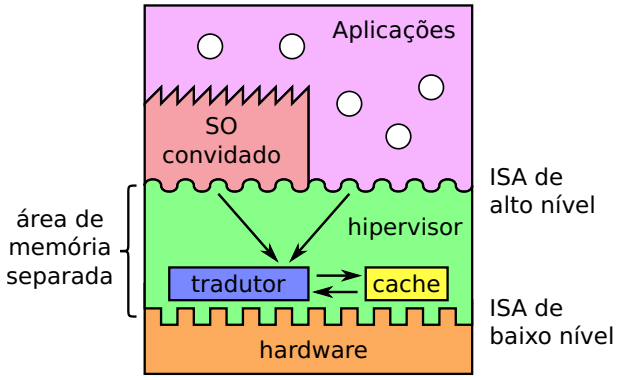

Figura 16: Hipervisor embutido no hardware.

Cabe lembrar que a virtualização da interface de sistema só pode ser aplicada diretamente caso o hardware subjacente atenda os requisitos de Goldberg e Popek (cf. Seção 2.1). No caso de processadores que não atendam esses requisitos, podem existir instruções sensíveis que executem sem gerar interrupções, impedindo o hipervisor de interceptá-las. Nesse caso, será necessário o emprego de técnicas complementares, como a *tradução dinâmica* das instruções sensíveis. Obviamente, quanto maior o número de instruções sensíveis, maior o volume de interpretação de código a realizar, e menor o desempenho da máquina virtual. Os processadores mais recentes das famílias Intel e AMD discutidos na Seção 2.2 atendem os requisitos de Goldberg/Popek, e por isso suportam esta técnica de virtualização.

Exemplos de sistemas que implementam esta técnica incluem os ambientes *VMware Workstation*, *VirtualBox*, *MS VirtualPC* e *KVM*.

4.3 Tradução dinâmica

Uma técnica frequentemente utilizada na construção de máquinas virtuais é a *tradução dinâmica* (*dynamic translation*) ou recompilação dinâmica (*dynamic recompilation*) de partes do código binário do sistemas convidado e suas aplicações. Nesta técnica, o hipervisor analisa, reorganiza e traduz as sequências de instruções emitidas pelo sistema convidado em novas sequências 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 sequências de instruções geradas pelo sistema convidado, de forma a melhorar o desempenho de sua execução. Neste último caso, os blocos de instruções muito frequentes 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 tradução dinâmica é composta basicamente dos seguintes passos [Ung and Cifuentes, 2006]:

1. *Desmontagem* (*disassembling*): o fluxo de bytes do código convidado em execução é decomposto em blocos de instruções. Cada bloco é normalmente composto de uma sequência 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 frequente 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 do processador real subjacente, o que torna desnecessário traduzir os blocos de instruções em uma representação independente de máquina.

4.4 Paravirtualização

A Seção 2.2 mostrou que as arquiteturas de alguns processadores, como o *Intel x86*, podem ser difíceis de virtualizar, porque algumas instruções sensíveis não podem ser interceptadas pelo hipervisor. Essas instruções sensíveis devem ser então detectadas e interpretadas pelo hipervisor, em tempo de carga do código na memória.

Além das instruções sensíveis em si, outros aspectos da interface software/hardware trazem dificuldades ao desenvolvimento de máquinas virtuais de sistema eficientes. Uma dessas áreas é o mecanismo de entrega e tratamento de interrupções pelo processador, baseado na noção de um *vetor de interrupções*, que contém uma função registrada para cada tipo de interrupção a tratar. Outra área da interface software/hardware que pode trazer dificuldades é a gerência de memória, pois o TLB (*Translation Lookaside Buffer*, Seção **??**) dos processadores *x86* é gerenciado diretamente pelo hardware, sem possibilidade de intervenção direta do hipervisor no caso de uma falta de página.

Em meados dos anos 2000, alguns pesquisadores investigaram a possibilidade de modificar a interface entre o hipervisor e os sistemas operacionais 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. Em particular, o hipervisor define uma API denominada *chamadas de hipervisor* (*hypercalls*), que cada sistema convidado deve usar para acessar a interface de sistema do hardware virtual. Todavia, a interface de usuário (*user ISA*) do hardware é preservada, permitindo que as aplicações convidadas executem sem necessidade de modificações. A Figura 17 ilustra esse conceito.

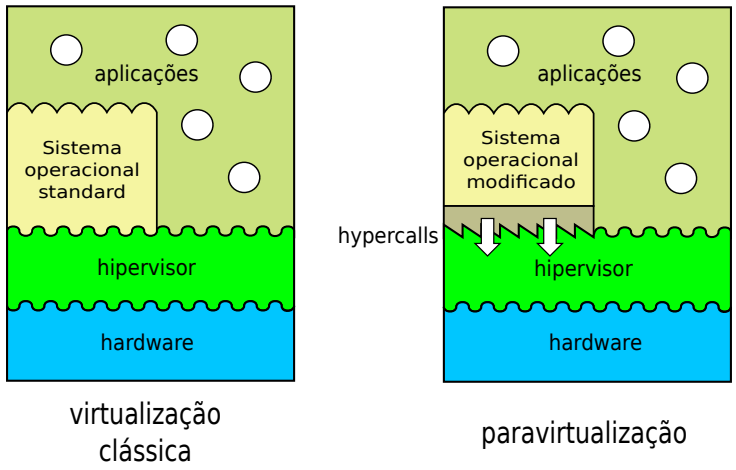

Figura 17: Paravirtualização.

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* (vide Seção 6.3) permite executar sistemas operacionais convencionais como Linux e Windows, modificados para executar sobre um 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 os conflitos, no caso de múltiplos sistemas convidados em execução simultânea. 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.

4.5 Aspectos 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 18):

  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 18):

  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 19, é usada pelo sistema *VMware* [VMware, 2000].

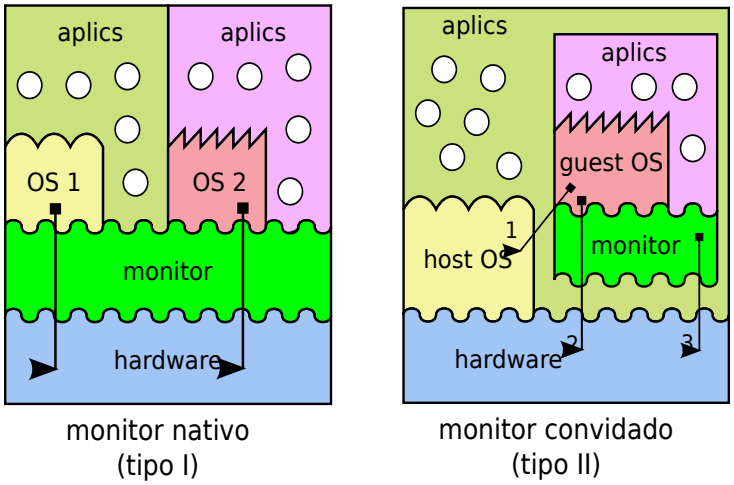

Figura 18: Otimizações em sistemas de máquinas virtuais.

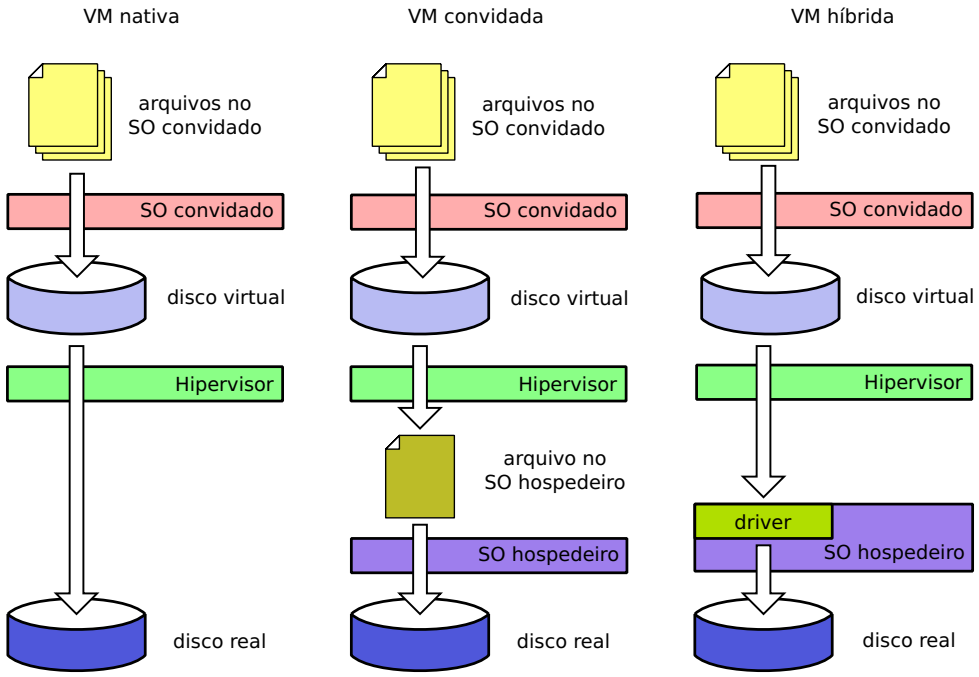

Figura 19: Desempenho de hipervisores nativos e convidados.

5 Aplicações da virtualização

Por permitir o acoplamento entre componentes de sistema com interfaces distintas, a virtualização tem um grande número de aplicações possíveis. As principais delas serão brevemente discutidas nesta seção.

O campo de aplicação mais conhecido da virtualização é a **portabilidade de aplicações binárias**. Este uso de máquinas virtuais começou na década de 1970, com o compilador UCSD Pascal. Esse compilador traduzia o código-fonte Pascal em um código binário *P-Code*, para uma máquina virtual chamada *P-Machine*. A execução do código binário ficava então a cargo de uma implementação da *P-Machine* sobre a máquina-alvo.

Para executar a aplicação em outra plataforma, bastava portar a implementação da *P-Machine*. Esse esquema foi posteriormente adotado pelas linguagens Java, C#, Perl e Python, entre outras, nas quais o código fonte é compilado em um código binário (*bytecode*) para uma máquina virtual específica. Assim, uma aplicação Java compilada em *bytecode* pode executar em qualquer plataforma onde uma implementação da máquina virtual Java (*JVM - Java Virtual Machine*) esteja disponível.

O **compartilhamento de hardware** é outro uso frequente da virtualização, por tornar possível executar simultaneamente vários sistemas operacionais, ou várias instâncias do mesmo sistema operacional, sobre a mesma plataforma de hardware. Uma área de aplicação dessa possibilidade é a chamada *consolidação de servidores*, que consiste em agrupar vários servidores de rede (web, e-mail, proxy, banco de dados, etc.) sobre o mesmo computador: ao invés de instalar vários computadores fisicamente isolados para abrigar cada um dos serviços, pode ser instalado um único computador, com maior capacidade, para suportar várias máquinas virtuais, cada uma abrigando um sistema operacional convidado e seu respectivo serviço de rede. Essa abordagem visa aproveitar melhor o hardware existente: como a distribuição dos recursos entre os sistemas convidados pode ser ajustada dinamicamente, pode-se alocar mais recursos aos serviços com maior demanda em um dado instante.

Além dessas aplicações, diversas outras possibilidades de aplicação de ambientes de máquinas virtuais podem ser encontradas, entre as quais:

Suporte a aplicações legadas : pode-se preservar ambientes virtuais para a execução de aplicações legadas, sem a necessidade de manter computadores reservados para isso.

Experimentação com redes e sistemas distribuídos : é possível construir uma rede de máquinas virtuais, comunicando por protocolos de rede como o TCP/IP, sobre um único computador hospedeiro. Isto torna possível o desenvolvimento e implantação de serviços de rede e de sistemas distribuídos sem a necessidade de uma rede real, o que é especialmente interessante dos pontos de vista didático e experimental.

Ensino : em disciplinas de rede e de sistema, um aluno deve ter a possibilidade de modificar as configurações da máquina para poder realizar seus experimentos. Essa possibilidade é uma verdadeira "dor de cabeça" para os administradores de laboratórios de ensino. Todavia, um aluno pode lançar uma máquina virtual e ter controle completo sobre ela, mesmo não tendo acesso às configurações da máquina real subjacente.

Segurança : a propriedade de isolamento provida pelo hipervisor torna esta abordagem útil para isolar domínios, usuários e/ou aplicações não-confiáveis. As máquinas virtuais de sistema operacional (Seção 3.2) foram criadas justamente com o objetivo de isolar sub-sistemas particularmente críticos, como servidores Web, DNS e de e-mail. Pode-se também usar máquinas virtuais como plataforma de execução de programas suspeitos, para inspecionar seu funcionamento e seus efeitos sobre o sistema operacional convidado.

Desenvolvimento de software de baixo nível : o uso de máquinas virtuais para o desenvolvimento de software de baixo nível, como partes do núcleo do sistema operacional, módulos e protocolos de rede, tem vários benefícios com o uso de máquinas virtuais. Por exemplo, o desenvolvimento e os testes podem ser feitos sobre a mesma plataforma. Outra vantagem visível é o menor tempo necessário para instalar e lançar um núcleo em uma máquina virtual, quando comparado a uma máquina real. Por fim, a execução em uma máquina virtual pode ser melhor acompanhada e depurada que a execução equivalente em uma máquina real.

Tolerância a faltas : muitos hipervisores oferecem suporte ao *checkpointing*, ou seja, à possibilidade de salvar o estado interno de uma máquina virtual e de poder restaurá-lo posteriormente. Com *checkpoints* periódicos, torna-se possível retornar a execução de uma máquina virtual a um estado salvo anteriormente, em caso de falhas ou incidentes de segurança.

Recentemente, a virtualização vem desempenhando um papel importante na gerência de sistemas computacionais corporativos, graças à facilidade de *migração* de máquinas virtuais implementada pelos hipervisores modernos. Na migração, uma máquina virtual e seu sistema convidado são transferidos de um hipervisor para outro, executando em equipamentos distintos, sem ter de reiniciá-los. 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 frequentes para implementar a migração de máquinas virtuais 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 do sistema operacional convidado 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.

6 Ambientes de máquinas virtuais

Esta seção apresenta alguns exemplos de 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.

6.1 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). O *VMware ESX Server* implementa ainda a paravirtualização. Por razões de desempenho, o hipervisor do *VMware* utiliza uma abordagem híbrida (Seção 4.5) 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.

6.2 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 3.2. 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 [McKusick and Neville-Neil, 2005].

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 de rede 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.

6.3 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. De acordo com seus desenvolvedores, 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 20, 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*.

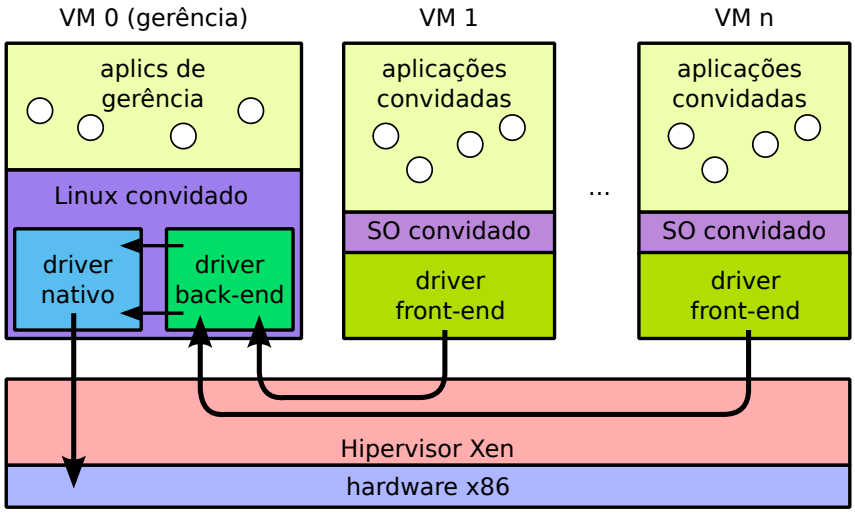

Figura 20: O hipervisor *Xen*.

6.4 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`[1]. Usando a chamada `ptrace`, o hipervisor recebe o controle de todas as chamadas de sistema de entrada/saída geradas pelas máquinas virtuais. Todos os sinais gerados ou

[1]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.

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.

6.5 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) 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*.

6.6 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 ou depuração de um programa.

6.7 JVM

É comum a implementação do suporte de execução de uma linguagem de programação usando uma máquina virtual. Um bom exemplo 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 21 mostra os principais componentes da plataforma Java.

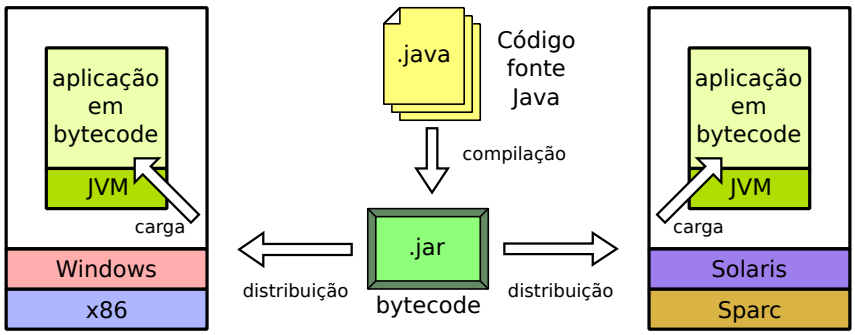

Figura 21: Máquina virtual Java.

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.

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 *ACM Symposium on Operating Systems Principles*, pages 164–177.

[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 *Symposium on Networked Systems Design and Implementation*.

[Corbató and Vyssotsky, 1965] Corbató, F. J. and Vyssotsky, V. A. (1965). Introduction and overview of the Multics system. In *AFIPS Conference Proceedings*, pages 185–196.

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

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

[Goldberg, 1973] Goldberg, R. (1973). Architecture of virtual machines. In *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.

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

[McKusick and Neville-Neil, 2005] McKusick, M. and Neville-Neil, G. (2005). *The Design and Implementation of the FreeBSD Operating System*. Pearson Education.

[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 *ACM Conference on Programming Language Design and Implementation*, 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.

[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.

[Price and Tucker, 2004] Price, D. and Tucker, A. (2004). Solaris zones: Operating system support for consolidating commercial workloads. In *18th USENIX conference on System administration*, pages 241–254.

[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 technology and future trends. *IEEE Computer*.

[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*.

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

[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 *USENIX Annual Technical Conference*, pages 1–14.

[Uhlig et al., 2005] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A., Martins, F., Anderson, 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). Denali: A scalable isolation kernel. In *ACM SIGOPS European Workshop*.

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