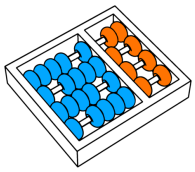
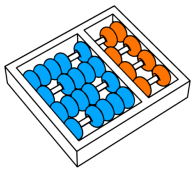


Alisson Linhares de Carvalho

“Suporte para execução de máquinas virtuais nativas”

CAMPINAS
2015



Universidade Estadual de Campinas
Instituto de Computação

Alisson Linhares de Carvalho

"Suporte para execução de máquinas virtuais nativas"

Orientador(a): **Prof. Dr. Edson Borin**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão da Dissertação apresentada à banca examinadora por Alisson Linhares de Carvalho, sob orientação de Prof. Dr. Edson Borin.

Assinatura do Orientador(a)

CAMPINAS
2015

Instituto de Computação
Universidade Estadual de Campinas

Suporte para execução de máquinas virtuais nativas

Alisson Linhares de Carvalho[1]

15 de maio de 2015

Banca Examinadora:

- Prof. Dr. Edson Borin (*Orientador*)
- Profa. Dra. Islene Calciolari Garcia - IC/UNICAMP
- Prof. Dr. Mario Antonio Ribeiro Dantas - INE/UFSC
- Prof. Dr. Sandro Rigo - IC/UNICAMP (*Suplente*)
- Prof. Dr. Alexandro José Baldassin - IGCE/UNESP (*Suplente*)

[1]Financial support: FAPESP scholarship (process 2013/26054-0) 2014–2015

Resumo

Uma máquina virtual é um programa de computador que emula uma interface para execução de outros programas. Esta tecnologia está presente em diversos sistemas computacionais e é utilizada desde o suporte a linguagens de programação de alto nível, como na máquina virtual Java, até a implementação de processadores com projeto integrado de *hardware* e *software*, como é o caso do processador Efficeon [60] da Transmeta.

Na tentativa de elevar o desempenho, alguns tipos específicos de máquinas virtuais podem requerer um acesso privilegiado a recursos administrados pelo sistema operacional. Consequentemente, em virtude dessa necessidade de obter um maior privilégio de execução, alguns projetos de máquinas virtuais optam por soluções nativas, removendo completamente a dependência de um sistema operacional. Essa solução, apesar da complexidade, possibilita a redução de grande parte da pilha de *software* existente entre a máquina virtual e o *hardware*, eliminando diversas restrições de acesso a dispositivos e a concorrência pela utilização de recursos.

Apesar da flexibilidade proporcionada pela adoção de uma arquitetura nativa, existe um preço a ser pago na escolha dessa abordagem. Implementar um *software* sem o auxílio das abstrações fornecidas por um sistema operacional é uma atividade complexa, suscetível a falhas e pode prejudicar gravemente a portabilidade. Dessa forma, conjecturamos que uma infraestrutura mínima, capaz de abstrair somente o essencial do *hardware*, fornecendo uma interface POSIX, simplificaria o projeto de novas máquinas virtuais nativas.

Assim, neste trabalho apresentamos o Native Kit, uma infraestrutura modular, compatível com o padrão POSIX e que foi construído para dar suporte à implementação e execução de máquinas virtuais nativas. Todos os módulos foram projetados de forma a eliminar o máximo de dependências, tornando esta infraestrutura simples, flexível e capaz de se adaptar aos diversos requisitos de projeto de máquinas virtuais nativas. Além disso, para validar e demonstrar como a nossa infraestrutura pode ser utilizado na prática, implementamos dois estudos de caso capazes de serem executados diretamente sobre o *hardware*, sem o auxílio de um sistema operacional. Esses estudos de caso poderão ser utilizados como ponto de partida para construção de novas máquinas virtuais nativas, abrindo espaço para diversas pesquisas relacionadas.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Este capítulo apresenta uma visão geral de alguns conceitos relevantes para a compreensão deste trabalho. Primeiramente, descrevemos as máquinas virtuais e suas principais aplicações. Em seguida, explicamos como o privilégio de acesso influência no funcionamento das máquinas virtuais. Por fim, apresentamos a nossa motivação para elaborar este projeto e o que esperamos obter com este trabalho.

O restante deste documento foi dividido da seguinte forma: o Capítulo 2 apresenta um resumo de alguns dos principais trabalhos relacionados; o Capítulo 3 descreve a infraestrutura desenvolvida neste projeto; o Capítulo 4 apresenta o suporte oferecido para o desenvolvimento de novas máquinas virtuais; o Capítulo 5 descreve como as aplicações podem fazer uso desta nova infraestrutura; por fim, o Capítulo 6 apresenta as conclusões obtidas neste trabalho.

1.1 Máquinas virtuais

Uma máquina virtual, como definida originalmente por Popek e Goldberg [73], é uma duplicata, isolada e eficiente, de uma máquina real. Em outras palavras, podemos descrever como um programa de computador capaz de emular uma interface para execução de outros programas.

Nos últimos anos, o termo máquina virtual ganhou uma elevada notoriedade, principalmente com o advento das linguagens híbridas[1], como C# e Java. Contudo, as máquinas virtuais são utilizadas em diversas áreas da computação. Desde o suporte para execução de programas em diferentes arquiteturas até a elaboração de processadores com projeto de *hardware* e *software* integrados.

[1] Termo utilizado para diferenciar as linguagens que, por padrão, trabalham com estágios de compilação e interpretação.

Na tentativa de organizar as máquinas virtuais de acordo com o funcionamento e aplicação, Smith e Nair dividem as máquinas virtuais em dois grupos: as máquinas virtuais de processo e as máquinas virtuais de sistemas [79].

As máquinas virtuais de processos são caracterizadas por proverem uma ABI virtual. Uma ABI, ou *Application Binary Interface*, é definida como uma interface entre dois ou mais *softwares* em uma mesma arquitetura. A ABI define como uma aplicação interage com o sistema operacional, as bibliotecas e com ela mesma. Além disso, ABI inclui todo o subconjunto de instruções não privilegiadas.

Diferente de uma máquina virtual de processo, onde somente a ABI é emulada, uma máquina virtual de sistema emula um sistema completo, incluindo toda a interface programável, o ISA[2]. Esse tipo de máquina virtual necessita de um gerenciamento avançado de recursos e, em alguns casos, a depender do proposito da máquina, pode haver duplicações de algumas operações realizadas pelo sistema operacional.

Na Figura 1.1 apresentamos um resumo didático dos principais tipos de máquinas virtuais existentes. Incluindo as máquinas virtuais de sistemas completos, que é o foco deste trabalho.

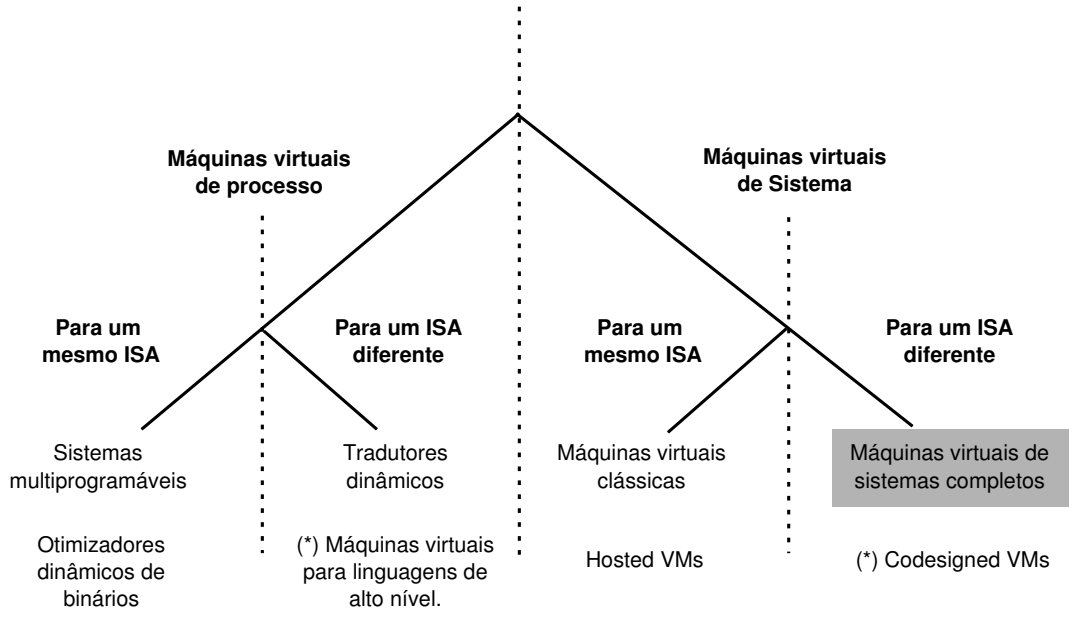


Figura 1.1: Taxonomia das máquinas virtuais.

De acordo com a taxonomia proposta por Smith e Nair, tanto as máquinas virtuais de processo como as máquinas de sistemas podem ser divididas com base no ISA. As

[2]Abreviação para *Instruction Set Architecture*.

máquinas virtuais que emulam o mesmo ISA podem compartilhar diversos recursos, como registradores, *flags* e instruções, sem a necessidade de grandes alterações na organização do código. Essa característica interfere diretamente no propósito e no funcionamento do sistema.

A seguir, descrevemos de forma sucinta os tipos de máquinas virtuais existentes - usando como base a taxonomia apresentada na Figura 1.1.

1.1.1 A multiprogramação

A multiprogramação é um recurso que permite que múltiplos processos possam ser executados de forma concorrente em um sistema operacional. Essa característica, comumente mencionada em livros de sistemas operacionais, pode ser categorizada como uma máquina virtual de processos.

Nos sistemas multiprogramáveis os processos são executados de forma concorrente e isolada, simulando a disponibilidade de um *hardware* completo e dedicado para cada aplicação. Dessa forma, podemos afirmar que na multiprogramação parte do *hardware* é virtualizado, já que cada processo pode acessar um subconjunto dos recursos de *hardware* diretamente, como é o caso da memória principal e das instruções de usuário.

1.1.2 Os emuladores e os tradutores dinâmicos de binários

Algumas máquinas virtuais são projetadas para dar suporte à execução de programas oriundos de outras arquiteturas. Nesse tipo de aplicação, em virtude da incompatibilidade entre os ISAs, existe a necessidade de emular o comportamento do conjunto instruções original.

Smith e Nair [79] definem emulação como o processo de implementar uma interface, incluindo as funcionalidades de um sistema, em um outro sistema.

A técnica mais simples de emulação é a interpretação. A interpretação é dividida em ciclos que envolvem o carregamento, a análise e a execução de cada instrução de um programa. Esses ciclos são feitos de forma contínua, executando uma instrução de cada vez. Em consequência disso, a interpretação é extremamente lenta e é utilizada na prática em parceria com outras técnicas mais eficientes.

As técnicas de prédecodificação e *direct threaded interpretation* [59, 48], por exemplo, pode ser usadas na prática para acelerar o processo de interpretação. No entanto, estas técnicas introduzem complicações, como a descoberta de código auto-modificável, que pode tornar a implementação extremamente complexa.

Uma alternativa mais eficiente que a pré decodificação é tradução, onde cada instrução do programa hóspede é substituída por um conjunto de instruções da máquina hospedeira [78], mantendo o mesmo comportamento do programa original. A tradução

pode ser estática, onde todo o código binário é traduzido antes da aplicação ser executada, ou dinâmica, traduzindo o código hospedeiro à medida que ele é executado.

Como exemplo de emuladores, podemos mencionar o Rosetta [4], Digital FX!32 [54] e o Intel IA-32 EL [36]. A ferramenta Rosetta permite que os usuários de computadores Mac, baseados na arquitetura x86, executem programas que foram compilados para a arquitetura PowerPC de forma transparente. Similar ao Rosetta, as ferramentas Digital FX!32 e Intel IA-32 EL são emuladores que permitem a execução de programas x86 em arquiteturas Alpha e Itanium, respectivamente.

1.1.3 Otimizadores e Instrumentadores dinâmicos

Os otimizadores dinâmicos são máquinas virtuais que trabalham de maneira similar aos tradutores dinâmicos, aplicando otimizações pontuais com base na execução dos programas. A ferramenta Dynamo [35], desenvolvida pela HP, por exemplo, é um otimizador capaz de melhorar o funcionamento de programas em tempo de execução.

Similar ao funcionamento dos otimizadores, que analisam características do funcionamento dos programas, temos as ferramentas de instrumentação. O Pin [65] e Valgrind [71], são exemplos de máquinas virtuais que analisam e perfilam programas durante a sua execução, com objetivo de extrair informações relevantes sobre o comportamento da aplicação.

1.1.4 Suporte para linguagens de alto nível

Um dos grandes problemas da computação moderna é a portabilidade. Os ambientes HLL[3] provêem recursos sofisticados que reduzem as diferenças existentes entre os milhares de configurações de *hardware* e *software*.

A plataforma Java [62], introduzida pela Sun Microsystems, e a Common Language Infrastructure [39], introduzida pela Microsoft, são exemplos de infraestruturas que fazem uso de implementações de máquinas virtuais para dar suporte à execução de programas em arquiteturas com conjunto de instruções diferente. Um outro exemplo recente de HLL é o V8, do google, uma *engine* moderna que faz uso de diversas técnicas de otimização, incluindo tradução dinâmica de binários para acelerar a execução da linguagem javascript [28].

Essa classe de máquinas virtuais também são aplicadas na área de sistemas operacionais, de forma transparente para o usuário, melhorando a compatibilidade dos programas entre as diferentes configurações de *hardware* e *software*. O Dalvik, por exemplo, é uma máquina virtual que está presente no sistema operacional *Android* [34]. O Dalvik permite

[3]Abreviação para *High Level Language Environments*.

que aplicativos, compilados para os *bytecodes* da máquina virtual, seja compatíveis com diferentes variações de arquiteturas e configurações. Recentemente, uma adaptação não oficial do *Android*, o Android-x86, provou que é possível instalar e executar programas compilados para *tablets* que utilizam o Android inclusive em computadores pessoais [3].

1.1.5 As máquinas virtuais clássicas

As máquinas virtuais clássicas são os tipos mais antigos de máquinas virtuais. São caracterizadas por prover um ambiente capaz de multiplexar o *hardware*, permitindo que múltiplos sistemas operacionais possam ser executados de forma concorrente. Atualmente, esse tipo de máquina virtual é utilizada com frequência em servidores, virtualizando recursos físicos.

A virtualização de processadores, memória e dispositivos de entrada e saída, por exemplo, para execução de múltiplos sistemas operacionais, permite um uso mais eficaz do *hardware* nativo, melhorando a segurança, escalabilidade e disponibilidade de recursos. Diversas máquinas utilizam dessa abordagem, como é o caso do Xen [8], Oracle VM [22], KVM [57] e o VMware Server [87].

Nessa classe de máquina virtual uma camada de *software* denominada monitor, ou do inglês *hypervisor*, passar a gerenciar e escalonar os recursos do *hardware* nativo. Isso permite que sistemas operacionais diferentes possam ser executados em um mesmo computador de forma transparente.

1.1.6 As máquinas virtuais hospedadas

Assim como as máquinas virtuais clássicas, as máquinas virtuais hospedadas são capazes de prover um ambiente para execução de sistemas operacionais completos. Contudo, diferente da abordagem clássica, as máquinas virtuais hospedadas são instaladas como programas dentro de um sistema operacional, simplificado o trabalho de instalação e configuração.

Atualmente, diversas ferramentas utilizam essa abordagem. A ferramenta Bochs [70], por exemplo, é uma máquina virtual hospedada que emula a arquitetura x86 utilizando a técnica de interpretação. Já o VirtualBox [29], VirtualPc [18], VMware Player [30] e o Parallels Desktop [24] são máquinas hospedadas eficientes, que fazem uso de virtualização com o auxílio do *hardware* para elevar o desempenho.

1.1.7 Co-designed VMs

As co-designed VMs[4] são máquinas virtuais com projeto integrado de *hardware* e *software*. Os processadores Crusoe [53] e Efficeon [60], da Transmeta, por exemplo, implementam a arquitetura x86 a partir de uma co-designed VM. Tanto o Efficeon como o Crusoe trabalham nativamente com um conjunto de instruções VLIW[5] proprietário. Uma camada de *software* [45] executada nativamente emula a arquitetura x86 utilizando-se de técnicas como interpretação e tradução dinâmica de binários. A tradução dinâmica é acompanhada de uma otimização dos binários, elevando o desempenho e reduzindo o consumo de energia.

1.1.8 As máquinas virtuais de sistemas completos

As máquinas virtuais de sistemas completos são caracterizadas por prover um ambiente capaz de emular um sistema em sua totalidade. Esse tipo de máquina virtual é bastante similar as máquinas virtuais clássicas e hospedadas, mas apresenta uma diferença marcante: a arquitetura da máquina emulada é diferente da arquitetura da máquina nativa. Desta forma, essa classe de máquinas virtuais trabalha com conjunto de instruções diferentes do *hardware* nativo.

Esse tipo de máquina virtual utilizado com bastante frequência para emular consoles de videogames, *hardwares* antigos e para prover retrocompatibilidade com dispositivos e funcionalidades antigas. Além disso, essas máquinas também são utilizadas para o desenvolvimento de sistemas onde a implementação do *hardware* é concomitante ao desenvolvimento do *software*. Nesses sistemas, uma máquina virtual permite a execução e teste de programas, sem que o *hardware* esteja completamente implementado.

O Qemu [21], por exemplo, é uma máquina virtual de sistema completo capaz de emular o comportamento de diversos ISAs diferentes, entre eles os ARM e o x86. O Qemu faz uso de tradução dinâmica de binários para acelerar o processo de emulação.

1.2 Privilégio de execução das máquinas virtuais de sistemas

Em virtude da natureza deste tipo de aplicação, alguns tipos específicos de máquinas virtuais, na tentativa de obter mais desempenho, podem requerer um acesso privilegiado a recursos administrados pelo sistema operacional. Com relação ao privilégio de acesso, Smith e Nair classificam as máquinas virtuais em três grupos [79], conforme ilustrado na Figura 1.2.

[4]Abreviação para *Virtual Machines*
[5]Abreviação para *Very-Long Instruction Word*.

As máquinas virtuais hospedadas são executadas como aplicativos em modo usuário, fazendo uso de todos os recursos fornecidos por um sistema operacional. Pode-se dizer, que está é uma forma conveniente para projetar uma máquina virtual, em virtude da elevada abstração do *hardware*, proporcionada principalmente pelos *drivers* e bibliotecas instaladas no sistema. Contudo, essa abstração tem um preço. O acesso ao *hardware* passa a ser limitado pelo sistema operacional, comprometendo o desempenho no acesso às funcionalidades privilegiadas. Uma alternativa para contornar esse problema, consiste na alteração de parte do sistema operacional, gerando uma arquitetura de privilégio híbrido, como ilustrado na Figura 1.2 (d). O KVM [57], por exemplo, é um tipo de máquina virtual híbrida que instala *drivers* e bibliotecas especializadas no sistema, obtendo acesso privilegiado a certos recurso, como os dispositivos de entrada e saída.

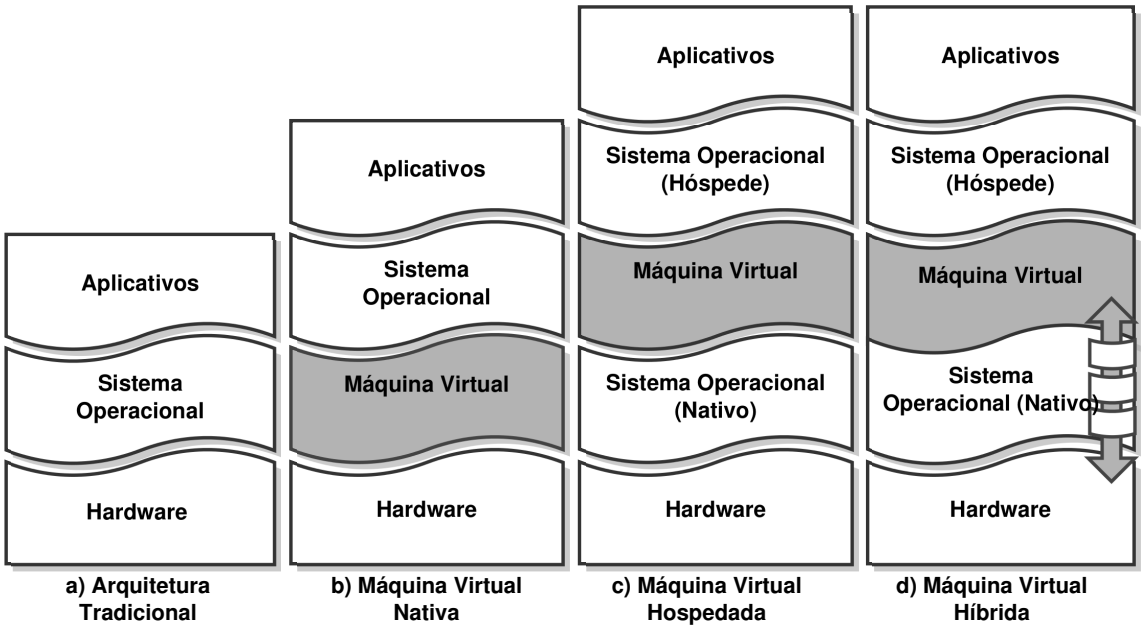


Figura 1.2: Privilégio de execução das máquinas virtuais.

Uma máquina virtual nativa é uma solução mais agressiva, quando comparada com uma máquina virtual híbrida. Na arquitetura nativa a máquina virtual tem acesso a instruções de nível privilegiado, executando diretamente sobre o *hardware*, sem o auxílio de um sistema operacional [79]. Por se tratar de um *software* nativo, existem diversos detalhes que devem ser gerenciados para que o funcionamento de todo o sistema seja garantido. O que torna essa abordagem complexa de ser projetada, visto que características existentes em alto nível, tais como gerenciamento de memória, gerenciamento de entrada e saída, gerenciamento de processos e *drivers* não existem sem a abstração proporcio-

nada pelo sistema operacional [80]. Apesar dessa dificuldade em se projetar este tipo de máquina, a eliminação de grande parte da pilha de *software* relacionada ao sistema operacional, bem como a concorrência por recursos entre processos não relacionados à aplicação, tornam o uso dessa abordagem altamente eficiente.

1.3 Motivação

No passado todo o trabalho de iniciar os dispositivos, configurar e carregar os programas era feito por operadores com auxílio de painéis e conectores. Escrever programas capazes de lidar diretamente com o *hardware* era algo natural, pois, em teoria, os operadores exerciam as tarefas atribuídas atualmente ao sistema operacional. Hoje, seis décadas depois, efetuar o mesmo trabalho de forma manual, em arquiteturas modernas, é algo relativamente complexo.

A utilização em massa dos sistemas operacionais simplificou consideravelmente o trabalho de escrever e executar programas de computador. Em consequência disso, o desenvolvimento de aplicativos nativos, capazes de serem executados sem o suporte de um sistema operacional, ficaram cada vez menos frequentes.

Apesar da dificuldade em projetar aplicações nativas, programas deste tipo são encontrados facilmente em sistemas embarcados, mais precisamente por questões relacionadas a desempenho e economia de energia. Estes equipamentos geralmente possuem baixa capacidade de processamento e espaço de memória. Por sua vez, exigem sistemas mais simples, se comparados aos computadores pessoais, que possuem um número muito maior de periféricos, controladoras e funcionalidades.

Na tentativa de simplificar a construção de aplicações nativas, foram criadas diversas bibliotecas. A NewLib, por exemplo, é uma biblioteca padronizada que pode ser facilmente portada para diversos dispositivos, necessitando apenas implementar um conjunto reduzido de chamadas de sistema [19]. Esta biblioteca apresenta um subconjunto das funções oferecidas pela biblioteca padrão do C/C++, porém adaptada para dispositivos embarcados.

A utilização de bibliotecas com suporte nativo trazem diversas vantagens, contudo isto não é o bastante. Todo o trabalho de escrever os *drivers* e gerenciar os recursos continua sob a responsabilidade do *software* nativo. Em alguns casos, podemos fazer uso de *frameworks* para desenvolvimento de sistemas operacionais para simplificar essa tarefa. Como exemplo, o OS Kit [51], um *framework* criado para simplificar a construção de sistemas operacionais. O OS Kit possui diversas ferramentas, *drivers* e módulos que podem ser utilizados na elaboração de aplicativos nativos, evitando retrabalho. Contudo, em algumas aplicações mais especializadas, como as máquinas virtuais nativas, esse tipo de *framework* pode conflitar com as necessidades das máquinas virtuais, tornando o

desenvolvimento mais restritivo.

1.4 Objetivo

Sabendo da dificuldade em se projetar máquinas virtuais nativas, este trabalho tem como objetivo principal propor uma infraestrutura para auxiliar o desenvolvimento de novas máquinas virtuais.

A arquitetura que estamos propondo permite que um *software*, uma vez compilado com a nossa infraestrutura, possa ser executado diretamente sobre o *hardware*, sem o auxílio do sistema operacional. Diversos recursos, como vídeo, teclado e disco, podem ser acessados quase que diretamente, eliminando grande parte da pilha de *software* necessária para efetuar as operações de entrada e saída.

Os módulos desenvolvidos foram planejados e projetados para gerenciar os recursos físicos de uma máquina virtual, podendo inclusive, serem modificados para se adaptar à carga de trabalho de um conjunto específico de aplicações. A depender dos requisitos dos programas que serão executados, os módulos não utilizados poderão ser suprimidos. Isso permite que a nossa infraestrutura forneça somente as funcionalidades essenciais, eliminando grande parte da pilha de *software* necessária para a execução de aplicativos hospedados.

Além disso, esperamos com este trabalho auxiliar na construção de diversos projetos desenvolvidos na UNICAMP, com ênfase no *Open ISA VM* - uma máquina virtual de sistema completo que está em estágios iniciais de desenvolvimento, no LMCAD (Laboratório Multidisciplinar de Computação de Alto Desempenho).

1.5 Contribuições

Apesar de o nosso foco ter sido o desenvolvimento de máquinas virtuais nativas, neste trabalho exploramos conceitos de diversas áreas da computação.

Na elaboração deste projeto foi necessário um estudo detalhado desde do desenvolvimento de aplicações nativas, até conceitos mais complexos, como o desenvolvimento de sistemas operacionais orientados à objeto. Por essa razão, acreditamos que este trabalho contribui em diferentes campos da computação e poderá ser utilizado em diversas pesquisas diferentes. Algumas das contribuições mais relevantes deste trabalho foram:

- Levantamento das principais máquinas virtuais que estão em funcionamento atualmente.
- Uma descrição detalhada do funcionamento de uma infraestrutura para suporte e desenvolvimento de máquinas virtuais nativas.

- Projeto e implementação de *drivers* e módulos que poderão ser utilizados em diversos projetos relacionados.
- Criação de um *framework* para auxiliar no teste de programas e módulos nativos.
- Automatização do processo de compilação e teste de aplicações nativas.
- Um resumo das principais características relacionadas ao desenvolvimento de aplicações nativas, com ênfase na família x86.
- Um resumo detalhado de como obter compatibilidade mínima com o padrão POSIX.
- Uma infraestrutura de código aberto, escrita de forma modular e completamente orientada a objeto.

Nos próximos capítulos discutiremos com mais detalhes o funcionamento da nossa infraestrutura e como ela pode ser utilizada no desenvolvimento de novas máquinas virtuais nativas.

Capítulo 2

Trabalhos relacionados

Como mencionado anteriormente, o propósito deste trabalho é prover uma infraestrutura capaz de auxiliar o desenvolvimento de máquinas virtuais nativas, dando ênfase no gerenciamento de recursos físicos.

Neste capítulo apresentamos uma visão geral do estado da arte no desenvolvimento de máquinas virtuais nativas. Além disso, descrevemos detalhadamente os principais trabalhos, encontrados na literatura, que se relacionam de alguma forma com este projeto.

Nas próximas seções discutiremos as principais máquinas virtuais nativas em uso atualmente, incluindo algumas abordagens relevantes ao tema apresentado. Em seguida, descrevemos o suporte em *hardware* existente nas duas principais famílias de processadores, AMD64 e ARM. Por fim, apresentamos uma visão geral, descrevendo onde este trabalho se encaixa em relação ao universo das máquinas virtuais.

2.1 Máquinas virtuais nativas

Existem diversos projetos de máquinas virtuais que utilizam uma arquitetura nativa com o objetivo de melhorar o desempenho e reduzir o desperdício de recursos. Grande parte dos projetos relacionados são destinados a servidores e são utilizados por empresas que oferecem recursos de computação nas nuvens[1]. Algumas dessas máquinas virtuais são bem conhecidas pela tecnologia empregada, é o caso do VMware Server 2 [87], Oracle VM Server [22] e Citrix XenServer [8].

Implementações de máquinas virtuais nativas também foram empregadas em projeto de sistemas operacionais escritos em Java. JNode OS e o JX são dois sistemas escritos, quase que inteiramente, em Java e apresentam implementações alternativas da JVM [12, 13]. Soluções com propósitos similares também são encontradas para sistemas

[1]Termo usado para classificar serviços computacionais providos remotamente, através da internet.

embarcados, tais como o Squawk [76] e o Xtrantum [44]. O Squawk é uma arquitetura projetada para executar programas Java em equipamentos de baixa capacidade de processamento, como sensores e dispositivos *wireless* [76, 77]. Diferente do Squawk o Xtrantum é uma solução baseada em um nano *kernel* que roda diretamente sobre o *hardware*. Foi projetado para executar diferentes sistemas operacionais, de forma concorrente, multiplexando os recursos computacionais. Objetivando uma melhor organização do projeto, sua arquitetura foi dividida em camadas. A HAL (*Hardware Abstraction Layer*) que implementa os *drivers* e é responsável por esconder a complexidade do *hardware* nativo, enquanto a ISL (*Internal Service Layer*) fornece um conjunto de bibliotecas que auxiliam no desenvolvimento das camadas superiores [67].

Um dos grandes problemas encontrados em máquinas virtuais nativas é o gerenciamento do *hardware*. Podemos afirmar que o sistema operacional, sob uma ótica *Top Down*, oferece uma perspectiva de um *hardware* mais simples, automatizando tarefas e tornando o desenvolvimento de *softwares* menos suscetível a falhas [66]. Como mencionado anteriormente, a ausência desse *software* exige que determinadas funções passem a ser gerenciadas pela máquina virtual, incluindo o gerenciamento dos *drivers*. Infelizmente não é possível gerenciar completamente todos os modelos de dispositivos para computadores pessoais comercializados, principalmente sem o auxílio dos fabricantes. A implementação de um *driver* para controladora ATA por exemplo, exige milhares de linhas de código [1]. Sem uma grande comunidade de desenvolvimento, a tarefa de escrever *drivers* se torna praticamente inviável, comprometendo, inclusive, a portabilidade, que atualmente é uma característica altamente desejável.

Na tentativa de reduzir o esforço necessário no gerenciamento do *hardware*, algumas máquinas virtuais fazem uso de *kernels* de propósito geral como parte de suas arquiteturas, simulando assim, o comportamento de um *software* nativo. Esses tipos de máquinas virtuais são denominados *Kernel Based Virtual Machines*, cujo principal representante dessa abordagem é o KVM. O KVM utiliza o Linux como parte de sua arquitetura buscando simplificar o desenvolvimento e compatibilidade com *drivers* e os *hardwares* mais modernos. Por fazer uso de um *kernel* bastante difundido, o KVM herda diversas características abstraídas pelo núcleo, tais como segurança e portabilidade [57].

Uma abordagem diferenciada no gerenciamento dos *drivers* foi apresentada pela Sun Microsystems na criação do Java OS [69]. Os *drivers* foram escritos em linguagem Java, implementando uma interface de comunicação com o *hardware*. Essa abordagem beneficiou a portabilidade ao criar uma separação entre os dispositivos e os *drivers*, porém essa solução não é boa o bastante. A interface dos *drivers* com a máquina real, continuará sendo necessária e todos os *drivers* ainda precisarão ser reescritos em Java. Outros sistemas mais recentes fizeram uso de abordagens similares, como é o caso do JNode OS, porém, sem grandes avanços [12].

Uma outra solução para gerenciamento de *drivers* pode ser encontrada no Xen. O Xen, é uma máquina virtual nativa que faz uso de paravirtualização. Ele separa os sistemas convidados em domínios. O domínio 0 é o único capaz de acessar o *hardware* diretamente. Neste domínio é executado uma versão modificada do Linux que é responsável por controlar as requisições enviadas pelo sistema operacional convidado, sequenciando o acesso ao *hardware* [43, 32]. Como o Linux possui uma interface bem definida e com diversos *drivers* disponíveis, o uso dessa abordagem traz como principal benefício uma maior portabilidade, separação de tarefas e simplifica o trabalho de escrever *drivers*. Infelizmente, essa solução gera uma sobrecarga no envio e recebimento de requisições dos domínios, inclusive no acesso ao *hardware* [64].

Uma outra alternativa, adotada por diversas máquinas virtuais para elevar o desempenho, é a criação de *drivers* especializados, compilados para máquina nativa [2, 57]. Contudo, essa abordagem apresenta um sério problema: uma dependência direta dos *drivers*. Caso a máquina virtual necessite ser executada em uma nova arquitetura, a depender das diferenças existentes, os *drivers* necessitarão ser recompilados ou, até mesmo, reescritos. De certa forma, isso pode ser considerado um limitante da portabilidade.

Uma alternativa elegante e que mantém todas as principais características existentes em uma abordagem de sistemas operacionais clássica, incluindo proteção de memória, gerenciamento de recursos e *drivers* sem perder os benefícios oferecidos por uma abordagem nativa, é propiciada pela arquitetura Exokernel [10]. Um Exokernel é um tipo de arquitetura onde o programa é integrado a uma biblioteca que auxilia no acesso direto ao *hardware*, reduzindo o impacto gerado pelas camadas de acesso contidas em um sistema operacional [47]. Uma arquitetura baseada em um Exokernel, por exemplo, pode ser especializada para um conjunto seleto de aplicações, e assim prover diversas características desejáveis em soluções nativas, tal como tolerância a falhas, mecanismos de depuração, inicialização de dispositivos e suporte a *drivers*.

Similar às técnicas adotadas pelo Xen e Java OS, uma alternativa que não é utilizada com frequência consiste em recompilar o sistema operacional hóspede, incluindo os *drivers*, para as instruções do processador virtual. Dessa forma, as execuções das instruções existentes nos *drivers* poderiam ser remapeadas para o *hardware* nativo, aproveitando o conteúdo do sistema operacional hóspede. Isso poderia trazer como principal benefício uma completa separação, entre os *drivers* e máquina virtual, elevando a portabilidade. Uma ideia parecida pode ser encontrada em sistema operacionais, mais precisamente na arquitetura Exokernel [10], onde múltiplos aplicativos podem se comunicar diretamente com o *hardware* e uma interface é responsável por multiplexar os recursos. Apesar do possível aumento na portabilidade da máquina virtual, essa solução pode comprometer a portabilidade do sistema hóspede, pelo fato dos seus *drivers* serem utilizados no controle da máquina real, computadores com dispositivos diferentes necessitaram de *drivers* apro-

priados. Contudo, esse problema pode ser facilmente resolvido com instalação de novos *drivers*, sempre que o sistema operacional hóspede for executado em um novo *hardware*.

2.2 Suporte em hardware para execução de máquinas virtuais

Nos últimos anos, em virtude da popularização das máquinas virtuais, tivemos a oportunidade de acompanhar um aumento considerável no suporte em *hardware* para virtualização e multiplexação de recursos. A Intel e a AMD, são as empresas que mais investiram nesse sentido, apresentado dezenas de extensões diferentes.

A Intel possui um padrão próprio de tecnologias para virtualização, denominado Virtualization Technology. Com o lançamento o Itanium, a Intel propôs duas extensões diferentes para cada arquitetura. O x86/AMD64 receberam o VT-x e o Itanium, destinado principalmente para servidores, recebeu o VT-i.

O VT-x e o VT-i contribuirão significativamente no desempenho das máquinas virtuais, trazendo como principais melhorias a inclusão de novos modos de operação, novos conjuntos de instruções para carga de tabelas de descritores (GDT, IDT, LDT, etc), suporte para compactação de páginas e virtualização de interrupções e exceções [83].

Similar à Intel, a AMD possui o AMD-V, sua própria extensão para virtualização. Existem diversas diferenças de implementação em relação ao VT-x, mas ambos possuem o mesmo objetivo: acelerar a execução de código virtualizado. Dentre as varias tecnologias presentes no AMD-V existe o RVI (*Rapid Virtualization Indexing*) uma tecnologia em *hardware* que permite a redução o *overhead* gerado quando a máquina virtual *guest* atualiza a tabela de páginas [46]. Em testes feitos pela VMware, a introdução dessa técnica gerou um ganho de desempenho de 42% em relação ao uso de *Shadow Pages* [38].

Mais recentemente, a Intel lançou o VT-d, acrônimo para *Virtualization Technology for Directed I/O*, que é atualmente suportado por diversas máquinas virtuais de sistemas. Essa tecnologia permite a criação de múltiplos domínios de proteção para acesso via DMA. Cada domínio é um ambiente isolado que contém um subconjunto da memória física. Com essa tecnologia, é possível associar um dispositivo a cada máquina virtual e, com o uso de domínios, ter um maior controle no acesso. Assim, essa tecnologia permite remapear o endereço DMA para um espaço de memória dentro da máquina virtual de forma direta, sem passar por camadas intermediarias [74, 40].

Na tentativa de elevar o desempenho do vídeo em máquinas virtuais, a Intel criou a tecnologia GVT, acrônimo de *Graphic Virtualization Technology*. A extensão GVT-d permite o acesso direto, de forma ilimitada, ao processador gráfico presente nas últimas CPUs da Intel. Essa tecnologia faz uso do suporte VT-d, permitindo que um *driver*

dedicado seja instalado no sistema operacional convidado.

Além do GVT-d a Intel dispõe de mais duas extensões a GVT-s e a GVT-g. A extensão GVT-g é destinada principalmente para servidores, onde é possível multiplexar o *hardware* gráfico completamente. O GVT-g usa uma camada em *software* que escalona o acesso ao *hardware* e protege os recursos privilegiados, enquanto um *driver* dedicado, instalado no sistema operacional convidado, tem acesso ao a recursos não críticos da placa de video. Experimentos com o gVirt[2], uma implementação aberta baseado no XenGT, demonstraram que é possível obter 95% do desempenho da GPU nativa e com ótima escalabilidade [82].

Já nos dispositivos móveis, o ARM v7 introduziu um o *Hypervisor execution mode*, permitindo a execução de código virtualizado em um nível de privilégio especial, similar ao apresente no Intel VT-x. Além disso, o ARM v7 possui um suporte para auxiliar no tratamento de interrupções e um novo nível de tradução de endereços para o DMA [27].

2.3 Sistemas operacionais especializados na execução de máquinas virtuais

Com base no propósito e na utilização, um sistema operacional pode ser categorizado em dois grupos distintos: sistema operacional de propósito geral e sistema operacional de propósito específico.

Os sistemas operacionais de propósito geral são caracterizados pela capacidade de executar e se adaptar a cargas de trabalho diferenciadas. Nos computadores domésticos, que na maioria dos casos utilizam este tipo de sistema, um usuário pode utilizar centenas de programas diferentes, de forma paralela, sem se preocupar com o comportamento das aplicações. Neste tipo de sistema, os desenvolvedores buscam um equilíbrio perfeito entre segurança, desempenho e usabilidade.

Diferente dos sistemas operacionais de propósito geral, utilizados principalmente em computadores domésticos e celulares, os sistemas operacionais de propósito específico são extremamente restritos e, na maioria dos casos, são projetados para um determinado dispositivo. São utilizados geralmente em situações onde o desempenho, segurança ou tempo de resposta devem ser priorizados.

Do ponto de vista prático, a maioria dos sistemas operacionais de propósito geral podem ser configurados para trabalhar de forma especializada. Em alguns casos é possível especializar o sistema operacional para um conjunto restrito de aplicações, removendo bibliotecas e recursos do *kernel* que não são utilizadas durante a execução dos aplicativos. Podemos, inclusive, reduzir o consumo de memória de uma forma mais agressiva, remover

[2]GVT-g é o nome comercial dessa tecnologia.

áreas de código do *kernel* que não são executadas durante o tempo de vida das aplicações, como por exemplo, algumas chamadas de sistemas [42].

O uso de sistemas operacionais especializados, pode melhorar o funcionamento das máquinas virtuais hospedadas, criando um *kernel* melhor adaptado à carga de trabalho gerada [56]. A depender dos requisitos da máquina virtual um sistema operacional especializado pode reduzir a pilha de *software* consideravelmente, fornecendo somente as funcionalidades requeridas pela aplicação. Consequentemente, isso pode contribuir para uma redução no número de trocas de contexto e no consumo de memória.

O OSv, por exemplo, é um sistema operacional otimizado para executar a máquina virtual Java. Uma das otimizações propostas pelo OSv parte da premissa de que como a JVM, durante a execução dos *bytecodes*, efetua validações de segurança evitando o acesso não autorizados a endereços fora da área dos programas, a JVM poderia ser executada em modo supervisor. Essa troca de prioridade dispensa as validações redundantes efetuadas pelo *kernel* e o custo na troca de modo de operação. Dessa forma, com o uso de um *kernel* especializado na execução de máquinas virtuais, o OSv consegue obter um elevado desempenho na execução de aplicativos Java [23, 58].

Na tentativa de simplificar o desenvolvimento de sistemas operacionais especializados, diversas bibliotecas foram criadas. Flux OS Toolkit [49], por exemplo, é uma biblioteca orientada a objeto, desenvolvida pela universidade de Utah e atualmente é utilizada em diversos projetos, incluindo o Fluke, um *kernel* que utiliza todos os recursos disponibilizados pelo OS ToolKit [51]. Uma das vantagens proporcionadas pelo OS Kit[3] é utilização de *drivers* oriundos do *FreeBSD* e do *Linux*, reduzindo o esforço no desenvolvimento de novos sistemas operacionais.

O OS Kit foi utilizado em diversos projetos relacionados ao desenvolvimento de máquinas virtuais, como a integração do Kaffe para criação de um ambiente Java nativo [49, 14] e na criação de máquinas virtuais recursivas [50].

2.4 O diferencial deste trabalho

Sabendo da necessidade de pesquisas na área de máquinas virtuais nativas, projetamos o Native Kit, uma infraestrutura responsável por dar suporte à execução de máquinas virtuais.

Similar ao funcionamento do OSv, uma máquina virtual integrada com a nossa infraestrutura pode ser configurada para obter acesso privilegiado a recursos do sistema, evitando validações de segurança redundantes. Contudo, a nossa arquitetura tem uma característica especial: ela foi concebida para executar diretamente sobre o *hardware* -

[3] Abreviação de Flux OS ToolKit.

diferente do OSv, que foi projetado para executar sobre um *hypervisor*. Dessa forma, a nossa infraestrutura pode ser comparada a uma extensão para o BIOS de um computador pessoal, já que fornecemos uma interface simples que abstrai e automatiza operações complexas no gerenciamento do *hardware*.

Assim como o OS Kit, o Native Kit foi projetado completamente de forma orientada à objeto, permitindo uma maior personalização e um reaproveitamento de código. Além, disso as duas infraestruturas provêem um suporte POSIX mínimo e um conjunto de componentes reutilizáveis.

Características	**OS Kit**	**Native Kit**
Arquiteturas compatíveis	x86, alpha	x86
Formato dos binários	ELF	ELF
Suporte para bibliotecas dinâmicas	sim	não
Infraestrutura portável	sim	sim
Infraestrutura modular	sim	sim
Infraestrutura orientado à objeto	sim	sim
Licença	GPLv2	GPLv3
Ano da última versão	2002	2015
Número de componentes	34	5
Segurança	Suporte variável	Suporte mínimo
Compatibilidade com POSIX	Suporte variável	Suporte mínimo
Gerenciamento de arquivos	Suporte variável	Suporte mínimo
Gerenciamento de memória	Suporte variável	Suporte mínimo
Gerenciamento de processos	Suporte básico	Suporte mínimo
Gerenciamento de *threads*	Suporte completo	Não
Gerenciamento de rede	Suporte variável	Não
Suporte para *drivers*	Suporte completo	Suporte mínimo
Depuração e testes	Suporte básico	Suporte básico
Complexidade	Média	Variável
Pilha de software	Média	Baixa

Tabela 2.1: Resumo comparativo entre o NK e o OS Kit.

Na Tabela 2.1 sumarizamos as principais funcionalidades deste trabalho e as comparamos com as características do OS Kit, que é o projetos que mais se aproxima da nossa proposta. Classificamos os recursos em quatro grupos diferentes. Um suporte mínimo significa que somente o fundamental para manter o sistema em funcionamento é oferecido; um suporte básico significa que o sistema provê alguns recursos extras que não são

fundamentais, mas que agregam valor ao projeto; um suporte completo é uma garantia de que a implementação é suficiente para a maioria das aplicações; por fim, um suporte variável significa que a infraestrutura pode prover funcionalidades básicas ou avançadas dependendo do requerimento do desenvolvedor da aplicação.

O OS kit é formado por um conjunto de módulos e componentes reutilizáveis, projetados para auxiliar na construção de sistemas operacionais. Essa característica permite que o desenvolvedor se concentre no desenvolvimento de funcionalidades mais avançadas, ao invés de gastar um precioso tempo construído funcionalidades comuns a todos os sistemas operacionais, tal como *drivers*, chamadas de sistemas e bibliotecas. Dessa forma, o OS Kit é personalizável, permitindo que o desenvolvedor escolha quais componentes ele gostaria de utilizar e quais componentes ele gostaria de reimplementar.

Alguns componentes possuem apenas os recursos básicos, enquanto outros implementam um suporte que pode variar do básico ao avançado. A interface POSIX, por exemplo, é implementada pela liboskit_posix e uma outra biblioteca C mapeia as chamadas de sistemas para esta biblioteca. Os aplicativos mais simples podem utilizar a liboskit_c, que é uma biblioteca com os recursos mínimos para desenvolvimento de programas [4]. Já os *softwares* mais complexos, e mais exigentes, podem fazer uso de uma biblioteca C extraída do FreeBSD, e dessa forma ter recursos mais avançados.

Além do fato do OS Kit ter um foco diferente da nossa proposta, uma outra diferença marcante é que a nossa abordagem é bem mais agressiva. O nosso objetivo é fornecer somente o conjunto mínimo de recursos, deixando para máquina virtual o gerenciamento das funcionalidades não fornecidas pela nossa infraestrutura. Além disso, nossa implementação dos componentes são bem mais simples e transparentes para o desenvolvedor. Um exemplo disso são os *drivers*. Enquanto o OS Kit implementa uma camada para dar suporte para *drivers* do Linux e do FreeBSD, adicionando um nível extra de abstração, a nossa infraestrutura fornece somente um conjunto mínimo de *drivers* para auxiliar no desenvolvimento, deixando a cargo da máquina virtual o acesso direto aos recursos físicos.

Além do OS Kit, existem outros projetos que compartilham algumas características em comum com este trabalho. Nos próximos capítulos apresentaremos a infraestrutura que projetamos e mencionaremos com mais detalhes as pesquisas mais relevantes na área de máquinas virtuais nativas.

[4] No Native Kit utilizamos a NewLib para executar a mesma tarefa, como será explicado no Capítulo 3.

Capítulo 3

Suporte à execução de máquinas virtuais

Conforme descrito anteriormente, uma máquina virtual é um tipo de aplicação que pode se beneficiar com a adoção de um sistema operacional especializado. Isso se deve ao fato das máquinas virtuais possuírem exigências que não são compatíveis com os requisitos da maioria das aplicações. Por exemplo, a máquina virtual pode solicitar a posse de recurso de entrada de saída para acelerar o seu desempenho ou acessar funções privilegiadas do sistema para evitar redundância de recursos. Essas operações, por padrão, não são liberadas em um sistema operacional convencional, pois isso poderia ocasionar brechas de segurança nas aplicações.

Neste capítulo apresentamos os detalhes técnicos por trás da infraestrutura responsável por dar suporte à execução de máquinas virtuais. Inicialmente apresentamos uma visão geral sobre a infraestrutura. Em seguida, descrevemos todos os módulos e serviços detalhadamente. Por fim, apresentamos uma visão geral de como a arquitetura pode ser utilizada na composição de máquinas virtuais nativas, hospedadas e híbridas.

3.1 Infraestrutura desenvolvida

Pensando nas operações realizadas por algumas das principais máquinas virtuais do mercado, projetamos uma arquitetura capaz de simplificar o desenvolvimento desse tipo de aplicação, quando não há o suporte provido por um sistema operacional. A arquitetura desenvolvida é composta por sete módulos básicos: gerenciador de processos, gerenciador de memória, gerenciador de arquivos, gerenciador de interrupções, sistema de entrada e saída, interface de chamada de sistemas e a camada *ARCH*. Na prática, para manter a legibilidade do código, alguns módulos tiveram que ser quebrados em várias classes e métodos. Contudo, o funcionamento e as trocas de dados respeitam essa organização.

Uma das disposições possíveis dos módulos pode ser visualizada na Figura 3.1. Nessa imagem, apresentamos uma máquina virtual sendo executada como um aplicativo em modo usuário, fazendo uso da interface POSIX para se comunicar com o *kernel*.

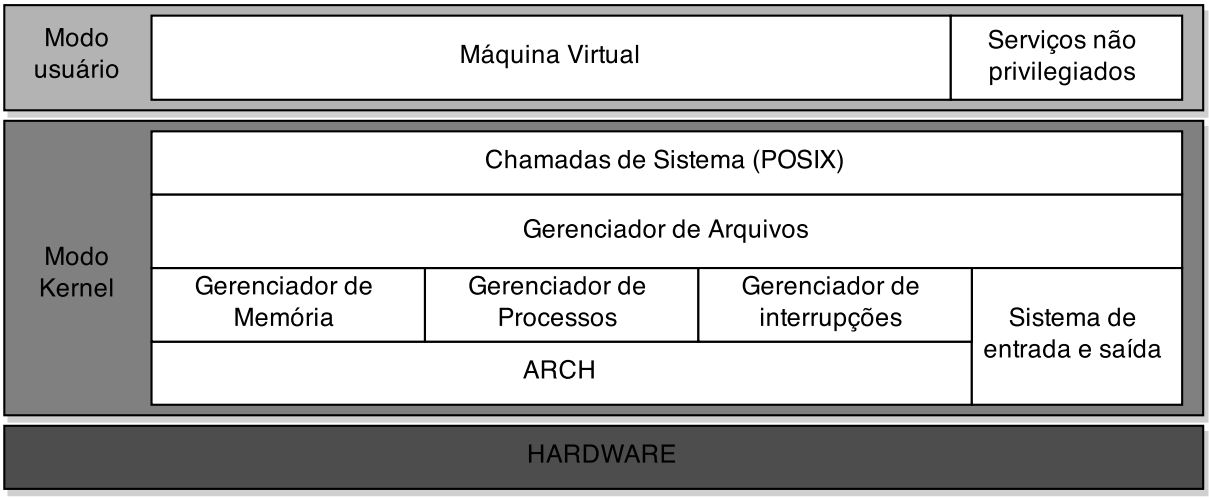


Figura 3.1: Arquitetura básica.

Em virtude da forma como os módulos foram projetados, é possível criar diversas variações da mesma arquitetura, mudando apenas o nível de privilégio da máquina virtual. Essa variação, apresentada na Figura 3.1, é a mais simples e didática - ideal para ser apresentada neste primeiro momento. Na Seção 3.8 exploraremos com mais detalhes as outras formas de organizar os módulos, incluindo as vantagens e desvantagens de cada abordagem.

Basicamente, todos os gerentes herdam da classe Manager, presente no arquivo Manager.h. Essa classe implementa as rotinas fundamentais que todos os gerentes devem possuir.

A união entre os módulos deve ser orquestrada. Um programa deve ser responsável por instanciar e fazer a união dos módulos. Por padrão, esse programa é o *kernel*. Nesse documento, para simplificar a compreensão, consideramos que o Native Kit é um *kernel* formado pela união de todos os módulos, mas na prática a implementação é bastante versátil. Caso um módulo não seja necessário, o mesmo poderá ser removido antes da compilação, permitindo, assim, diferentes configurações de um mesmo *kernel*.

Uma vez compilado o *kernel*, o nosso conjunto de ferramentas gerará uma imagem nativa do sistema. A imagem gerada é compatível com o padrão *multiboot* e pode ser carregada por qualquer *loader* que implemente este protocolo, incluindo o GRUB.

A nossa arquitetura permite que o programador da máquina virtual possa fazer uso de recursos de orientação a objeto, como polimorfismo e herança, para remover e adicionar

novas funcionalidades. Para tanto, todos os módulos possuem uma interface bem definida, que deve ser implementada. A interface entre os módulos garante a correta troca de mensagens entre os objetos, permitindo um elevado poder de personalização. Nas próximas seções discutiremos esta interface com mais detalhes, bem como as peculiaridades de cada módulo.

3.2 Gerenciamento de memória

O gerenciador de memória é um dos componentes mais importantes existentes em um sistema operacional. Sua principal tarefa é abstrair o acesso à memória, simplificado o trabalho de alocação e desalocação de dados. Na prática, este gerente pode apresentar ainda diversas outras atribuições, dependendo do propósito do sistema. Dentre as tarefas comumente atribuídas a este gerente, podemos mencionar: alocar e desalocar áreas de memória; controlar o tipo de proteção e acesso de cada região alocada; esconder detalhes físicos da *RAM*, de forma a abstrair características da arquitetura; detectar os módulos de memória conectados; calcular o tamanho de memória disponível e gerenciar a memória virtual.

Na tentativa de aumentar a portabilidade do sistema, incluindo todos os serviços construídos sobre a nossa infraestrutura, o módulo de gerenciamento de memória foi dividido em dois níveis: gerenciamento físico e gerenciamento virtual.

O gerenciador de memória física é um dos primeiros serviços a serem executados. Esta camada é responsável por detectar a quantidade memória disponível no sistema e abstrair detalhes da arquitetura a qual o sistema foi compilado. Esta camada apresenta uma interface organizada e conveniente, simplificando o acesso à memória e a portabilidade do sistema.

Já a camada de gerenciamento de memória virtual está diretamente ligada ao funcionamento dos processos e serviços presentes na arquitetura. Seu papel é muito importante para o sistema, pois é este o módulo responsável por controlar o endereçamento lógico dos processos e serviços. Além disso, sua principal funcionalidade nesta arquitetura é a de isolar as áreas de memória, prevenindo assim, o acesso a endereços não mapeados.

Nas próximas seções discutiremos os detalhes técnicos e problemas encontrados na elaboração das camadas de gerenciamento de memória, bem como o relacionamento existente entre o *hardware* e o *kernel*.

3.2.1 Memória física

Um dos grandes problemas em portar um sistema operacional para diversas arquiteturas é o fato dos computadores apresentarem diferenças marcantes na forma como a memória

física é apresentada. Na arquitetura PIC, por exemplo, que é uma família de micro controladores baseados na arquitetura *Harvard*, a memória de dados é separada da memória que contém o programa, e esta, por sua vez é apresentada na forma de pequenos bancos, sendo necessário informar previamente qual banco de memória será acessado em um determinado momento. Algo completamente diferente da arquitetura de memória utilizada em sistemas com processadores da família x86 ou ARM, em que os dados e códigos são armazenados na mesma memória e são acessados de forma contínua.

Além da inconveniência gerada pela diferença na forma de acessar a memória, em algumas arquiteturas, diversos endereços físicos são mapeados para periféricos. Alguns dispositivos, como as placas de vídeo podem reservar grandes regiões dos endereços de memória. Por exemplo, na família x86, por padrão, os endereços 0xB8000 até o endereço 0xB8C80 são reservados para o modo texto 80x40. Qualquer tentativa de escrever um dado nesta faixa de endereços provoca uma atualização do vídeo, tornando esta faixa obviamente inadequada para armazenamento de dados. Em modo gráfico, uma porção maior dos endereços físicos passam a ser remapeados para endereços da placa de vídeo. Se levarmos em consideração uma placa de vídeo *on-board*, que faz uso da memória principal para armazenar as informações gráficas, o espaço reservado para vídeo passa à ser proporcional à memória alocada: quanto maior a resolução configurada, mais endereços deverão ser reservados. Fazendo com que, na prática, a quantidade de memória disponível para os programas seja menor que a quantidade de memória física instalada.

A existência de áreas de memória compartilhadas e endereços reservados por dispositivos pode provocar pequenos inconvenientes na utilização da RAM, pois dados e códigos não podem ser alocados em qualquer endereço físico, tornando necessário um gerenciamento de quais endereços podem ou não ser utilizados pelo *loader* - o módulo responsável por carregar os binários para memória.

Na Tabela 3.1 podemos visualizar o mapa de memória retornado pela função E820 do *BIOS*[1], em um computador com arquitetura x86. Os endereços marcados com 2 são reservados exclusivamente para dispositivos e não podem ser utilizados para alocar dados. Já os endereços marcados com 1, representam as áreas que podem ser utilizados livremente, desde que os dados contidos nessas regiões não sejam mais necessários para o funcionamento do *hardware*. Além disso, apesar da padronização da função E820, computadores diferentes podem apresentar áreas de memória maiores, menores ou, até mesmo, mais regiões reservadas, tornando qualquer suposição do estado inicial de memória algo arriscado.

[1] A função getMemoryMap foi utilizada na geração desta tabela. O código desta funções podem ser encontrado no arquivo memory.cpp.

Endereço base	Tamanho	Tipo
0x00000000	0x0009FC00	1
0x0009FC00	0x00000400	2
0x000F0000	0x00010000	2
0x00100000	0x1FEFE000	1
0x1FFFE000	0x00002000	2
0xFFFC0000	0x00040000	2

Tabela 3.1: Mapa de memória retornado pelo BIOS.

Como podemos notar, o espaço de endereçamento físico mapeado para memória RAM nem sempre é contínuo, apresentando diversos detalhes que dificultam a sua utilização. O sistema operacional fornece uma camada de abstração da memória física que faz com que os programas sejam capazes de visualizar a memória de uma forma mais simples, simplificando o trabalho de programação.

Endereço	Tamanho	Descrição
0x00000000	0x000003FF	IVT - Interrupt Vector Table
0x00000400	0x000000FF	BDA - BIOS Data Area
0x00000500	0x0009F6FF	Memória convencional
0x00007C00	0x000000FF	Entry point
0x0009FC00	0x000003FF	EBDA - Extended BIOS Data Area
0x000A0000	0x0001FFFF	VGA Frame Buffers
0x000C0000	0x00007FFF	VBIOS - Video BIOS
0x000C8000	0x00027FFF	Memória livre para uso
0x000F0000	0x0000FFFF	BIOS
0x10000000	0xEEBFFFFF	Memória estendida
0xFEC00000	0x013FFFFF	Memória usada por diversos dispositivos

Tabela 3.2: Descrição da memória.

Em sistemas com processadores da família x86, a memória é dividida em duas áreas, em virtude da necessidade de manter compatibilidade com arquiteturas de legado. Na Tabela 3.2 o mapa de memória, apresentado anteriormente, foi expandido, mostrando as áreas padronizadas de memória. Essas áreas de memória são dividas em Memória Convencional e Memória Estendida. A Memória convencional é padronizada, consiste do primeiro *mega byte* de RAM. Já a Memória Estendida pode variar drasticamente, sendo necessário fazer uso do *BIOS* para determinar os endereços corretos de diversos

componentes. Assim, o gerenciador de memória deve utilizar as informações repassadas por camadas de baixo nível e construir um mapa de memória consistente, retornando para camadas superiores apenas os endereços que podem ser acessados. Além disso, o gerenciador de memória necessita detectar e manter os endereços reservados para alguns dispositivos importantes, como é o caso do vídeo.

Diferente da arquitetura x86, onde geralmente os fabricantes de placas mães seguem uma série de especificações e padrões preestabelecidos, algumas arquiteturas não compactuam das mesmas facilidades. Sistemas com processadores da família de arquiteturas ARM, por exemplo, não possui uma maneira padronizada de se obter informações sobre a memória. Em sistemas embarcados que utilizam ARM é comum as especificações dos endereços serem inseridas diretamente no código de *drivers* ou, em alguns casos, até mesmo no código do *kernel*. Pelo fato do ARM ser vastamente utilizado, os sistemas operacionais são bastante dependentes de configurações dos fabricantes, desta forma existe uma grande gama de dispositivos e *drivers* diferentes. Portanto, encontrar uma maneirar de obter compatibilidade com todos esses dispositivos é algo muito complexo de ser alcançado. Um mecanismo de gerenciamento portável tem que de alguma maneira abstrair a detecção de memória, para que o sistema funcione corretamente em diversas arquiteturas.

Em alguns casos é possível obter as configurações mínimas de memória através da técnica de *Memory Probing*, que consiste em tentativas sucessivas de leitura e escrita em todos os endereços de memória, na expectativa de receber exceções nos acessos a áreas inválidas. Contudo, está técnica é raramente usada, por não apresentar um mapa consistente de memória.

Na tentativa de elevar a portabilidade do sistema, fizemos uso do padrão *multiboot*. Desta forma, o mapa de memória física é construído com base nos dados retornados pelo *bootloader* retirando, desta forma, a obrigação do *kernel* de detectar a memória diretamente. O mapa retornado por um *bootloader* que suporta o padrão *multiboot* é bem similar ao apresentado pela função E820, contudo a forma de obtê-lo é extremamente simples, se comparado com a obtenção direta pelo *BIOS*.

3.2.2 Memória virtual

Em um sistema operacional, é comum a existência de dezenas e até centenas de processos sendo executados de forma concorrente. A memória virtual exerce um papel fundamental no funcionamento desse tipo de sistema, pois esta é responsável por permitir que múltiplos programas fiquem parcialmente residentes em memória, permanecendo apenas trechos importantes dos binários a cada execução. Além disso, a memória virtual simplifica o compartilhamento de bibliotecas dinâmicas e auxilia na construção de mecanismo de segurança, tais como isolamento de endereços e permissões de leitura, escrita e execução

em páginas.

Em uma máquina virtual nativa, a existência de múltiplos processos ou serviços sendo executados simultaneamente em espaços físicos diferentes é uma característica desejável, por exemplo: um processo responsável por depurar o *kernel* poderia rodar concorrentemente com a máquina virtual; as heurísticas de tradução poderiam ser atribuídas a um serviço concorrente com a emulação; ou o tradutor dinâmico de binários poderia ser um processo separado. Existem milhares de operações que podem ser divididas em espaços de endereços diferentes, com o propósito de auxiliar tanto no desempenho como no gerenciamento de uma máquina virtual. Assim, a principal tarefa do gerente de memória virtual, em nossa arquitetura, é de mapear os endereços de páginas virtuais para as regiões retornadas pela camada de gerenciamento de memória física.

Em determinado momento, o sistema operacional *guest* pode necessitar efetuar um *swap* ou um mapeamento para um endereço físico específico. Essas operações são exemplos de ações que deveriam ser gerenciadas pela máquina virtual, com objetivo de evitar que um endereço não disponível seja acessado de forma ilegal. Sabendo disso, é interessante permitir que, em determinadas situações, ações executadas pelo *kernel*, tal como um tratamento de uma falta de página ou uma falha de proteção geral possam ser propagadas e tratadas pela máquina virtual.

O gerenciamento de memória de uma máquina virtual nativa deve ser bastante dinâmico, permitindo que módulos com permissões de acesso registrem rotinas de tratamento de exceções e interrupções, de forma simples e eficiente. Além disso, como mencionando anteriormente, no gerenciamento de memória física, algumas áreas da RAM podem ser reservadas para dispositivos, como é o caso do vídeo. Desta forma uma tentativa de acesso a um endereço de vídeo, deve antes estar corretamente mapeada para o endereço físico correspondente. Um processo que necessite escrever diretamente em um componente de *hardware* mapeado em memória, deve primeiramente configurar o endereço virtual para uma área física consistente. Ter este tipo de acesso de uma forma conveniente e configurável por uma máquina virtual pode elevar o desempenho do sistema *guest* de forma considerável, principalmente pela possibilidade de acessar dispositivos diretamente, com o uso de técnicas baseadas em *passthrough*[61].

Infelizmente, construir um mecanismo de gerenciamento virtual portável não é uma tarefa trivial, pois esse tipo de mecanismo está diretamente ligado ao padrão usado na representação do vetor de páginas e do funcionamento do sistema de exceções do processador. A família de arquiteturas x86, por exemplo, utiliza uma técnica de mapeamento de páginas em dois níveis. O primeiro nível de páginas é controlado por um diretório, que está limitado a 1024 entradas de 4 *bytes*, quando o processador está operando em 32 *bits*. Cada entrada no diretório aponta para outro vetor de páginas com 1024 entradas de 4 *bytes*. Que por sua vez, possui ponteiros de 20 *bits* para os endereços físicos presentes

na RAM, 10 *bits* de *status* e 2 *bits* de permissão. Desta forma cada posição no diretório de páginas mapeia exatamente 4 MB de RAM. Com essa característica, o x86 consegue mapear 4 gb de endereçamento virtual quando o modo de operação é de 32 *bits*.

A paginação em dois níveis adotada pela família x86 é bastante eficiente, pois permite uma economia considerável de espaço de memória para armazenar as tabelas de tradução de endereços, já que não é necessário manter todo vetor de páginas preenchido, necessitando apenas o preenchimento das entradas que estiverem em uso pelo processo. Além disso, é bastante conveniente para o programador, pois a tabela de páginas pode ser acessada e modificada diretamente, desde que o processo esteja autorizado.

Tirando o modo padrão, o x86 suporta mais dois modos de operação: *super pages*, que elimina a necessidade de possuir um vetor de páginas, fazendo com que o diretório contenha ponteiros para blocos de 4 MB; e um suporte a *PAE*, que permite expandir o limite da memória física presente no sistema.

Já a arquitetura *ARM*, segue um modelo bastante similar ao x86. Contudo, apresenta diversas diferenças em relação aos tamanhos dos vetores, na quantidade de modos de operação e nas configurações.

A *MMU* do *ARM* suporta quatro tamanhos de páginas. A maior página tem o tamanho de 16 MB e é denominada *Super Section Page* e a menor é de 4 KB e é denominada *Small Page*. Alguns processadores *ARM* possui uma página ainda menor, a *Tiny*, mas essa, de acordo com o manual, foi removida no ARMv7. Além disso, o uso de páginas de 4 KB não é recomendável, existindo apenas por questões de compatibilidade. Estudos recentes comprovam que páginas de 16 KB, apesar de provocarem um pequeno aumento da fragmentação interna reduzem o número de TLB misses [11, 85].

Similar ao x86, o ARM possui uma hierarquia de dois níveis de páginas. Cada entrada na primeira tabela contém um ponteiro para um segundo nível, ou um ponteiro para uma seção que pode ser configurada para ter o tamanho de 1 MB ou 16 MB. O primeiro nível de páginas possui 16 KB e cada entrada é capaz de mapear 1 MB de RAM. Se o sistema estiver configurado para trabalhar com páginas de 4 KB, o segundo nível será uma tabela de 1024 entradas de 4 *bytes*, similar ao padrão adotado pelo x86. Além disso, da mesma forma que o x86 utiliza o registrador cr3 no carregamento do diretório de páginas, o primeiro nível da tabela de páginas no ARM deve ser armazenado no registrador TTBR1, de forma equivalente.

Visando elevar a portabilidade do gerenciamento de memória - mantendo pelo menos uma compatibilidade entre o x86 e ARM - o sistema de paginação do processador foi delegado para interface *ARCH*, que é responsável por controlar a proteção das páginas e interagir com as estruturas de mapeamento do processador. O *ARCH* possui uma série de rotinas minimalista que tentam obter o máximo de compatibilidade entre os processadores ARM e x86.

A comunicação entre a camada de gerenciamento de memória virtual com a interface *ARCH* é feita pelo compartilhamento de um vetor de 1024 entradas de 4 *bytes*. No vetor de páginas, cada entrada contém um ponteiro de 20 *bits* para uma página virtual de 4 KB e 12 *bits* de configuração. Os 3 *bits* mais importantes são: bit 0, que informa se a página é valida, bit 1 determina permissão de leitura e escrita e o bit 2 que determina a liberação de execução. Acompanhado com o vetor, é passado como referência o *offset* da região virtual a qual o vetor de páginas representa, que determina o deslocamento em múltiplos de 4 kB na memória. Desta forma, por ser bem parecido com segundo nível da tabela de páginas, tanto do x86 como do ARM, converter essa estrutura para o padrão nativo não será uma tarefa complexa no tocante à portabilidade. Contudo, é esperado que o seu desempenho seja maior na arquitetura x86, visto que em processadores diferentes pode haver um pequeno *overhead* na conversão da estrutura passada pela camada de gerenciamento virtual. Dessa forma, para elevar o desempenho em outras arquiteturas, pode ser necessário modificar a interface entre o *ARCH* e o sistema de memória virtual e configurar o tamanho das páginas para valores diferentes de 4 KB.

As rotinas de exceções, que detectam falhas de segurança e faltas de páginas, foram abstraídas de forma a tornar o gerenciamento de memória virtual completamente independente do processador. Toda configuração e implementação da interface de exceções foi atribuída à interface *ARCH*, que fornece uma camada orientada a eventos, onde é possível registrar dezenas de funções de tratamento diferentes. O modelo adotado foi inspirado em Javascript, onde é possível registrar tratadores para os principais eventos de uma forma simples e prática.

A camada *ARCH* é extremamente simples, mas requer um atenção especial. Desta forma, os detalhes de implementação, bem como as principais rotinas de tratamento de exceções e controle de páginas serão discutidas com mais detalhes na seção *ARCH*.

3.2.3 Alocação de páginas

No projeto do gerenciador de memória optamos por utilizar um alocador baseado em uma lista sequencial de blocos livres, usando a política *First Fit*. Essa política de gerenciamento de memória - podemos incluir centenas de variações - já foi exaustivamente discutida em diversos artigos [37, 81, 86]. Contudo, até onde sabemos, a nossa implementação é original e levemente diferente. Introduzimos algumas pequenas melhorias baseadas em características do gerenciamento de memória do Linux e em suposições a respeito das necessidades de uma possível máquina virtual implementada sobre a nossa infraestrutura.

No algoritmo de alocação, a memória disponível é dividida em blocos de páginas onde cada bloco de memória livre faz parte de uma lista encadeada. A nossa implementação parte da expectativa de que as áreas de memória serão fixas a maior parte do tempo, pois

a quantidade de processos criados e destruídos pelo *hypervisor* será extremamente baixa - quando comparado com um sistema operacional de propósito geral. Assim, esperamos que a maior parte do tempo, somente serviços da máquina virtual estarão em execução, e esta, por sua vez, ocupará praticamente toda a memória disponível. Desta forma, é interessante que o algoritmo tenha como principal foco um baixo consumo de memória, já que a sua principal responsabilidade é a de evitar que uma página inválida, ou previamente alocada, seja liberada para leitura ou escrita de dados.

O funcionamento do algoritmo proposto é extremamente intuitivo. Utilizamos na nossa implementação uma lista ordenada pelo endereço do bloco livre. Sempre que é necessário alocar uma página para algum processo, o *kernel* requisita uma quantidade de páginas física do gerente de memória, que por sua vez retira da lista encadeada de blocos livres uma quantidade de páginas suficiente para armazenar os dados do processo. Dessa forma, um endereço que não pode ser mapeado não será retornado pelo gerente de memória, tornando assim, as áreas reservadas invisíveis para as camadas superiores do *kernel*.

Sempre que um bloco de páginas é desalocado, ele é novamente inserido na lista de páginas livres, mas existe um detalhe: quando as páginas são apresentadas de forma contínua e consecutivas, isto é, sem intervalos entre as áreas, é feito uma união entre os blocos. Está união reduz a quantidade de ponteiros entre blocos e eleva o desempenho do alocador, já que a união reduz a quantidade de elementos na lista encadeada.

O algoritmo apresenta um baixo consumo de memória. Isto é conseguido graças à utilização das regiões disponíveis para armazenamento de informações pertinentes aos blocos livres. Essa característica faz com que o gerente de memória física ocupe logicamente[2] apenas quatro palavras de memória. O suficiente para manter um ponteiro para o primeiro elemento livre de um bloco e alguns dados estatísticos relevantes. No pior caso, em uma situação de fragmentação extrema, o algoritmo continuará usando apenas a mesma quantidade de memória lógica. Tornando esse algoritmo extremamente econômico, se comparado com outras variações. Desta forma, essa técnica é extremamente simples de ser mantida e apresenta um baixo consumo de memória, ou seja, disponibiliza um maior espaço físico para os processos.

Na Figura 3.2 podemos visualizar a operação de alocação e desalocação de uma página física. Na alocação, o algoritmo necessita varrer a lista de blocos livres na tentativa de encontrar um bloco capaz de atender a necessidade de espaço de um programa. No melhor caso, o algoritmo se comporta em O(1), sempre retornando o primeiro bloco de memória. Já em um caso extremo, o algoritmo seria em ordem O(n), necessitando varrer toda a lista de memória para para retornar um bloco livre. À primeira vista, esse algoritmo

[2]A memória lógica está relacionada a quantidade de memória disponível para processos e camadas superiores do *kernel*.

parece ineficiente. Contudo, na prática, o número de interações no pior caso, ou seja, o número máximo de interações para encontrar um bloco de páginas, é baseado no tamanho da memória disponível e do nível de fragmentação encontrado. O nível de fragmentação máximo é atingido quando existe uma alternância entre uma página alocada e outra disponível - é algo bem difícil de acontecer.

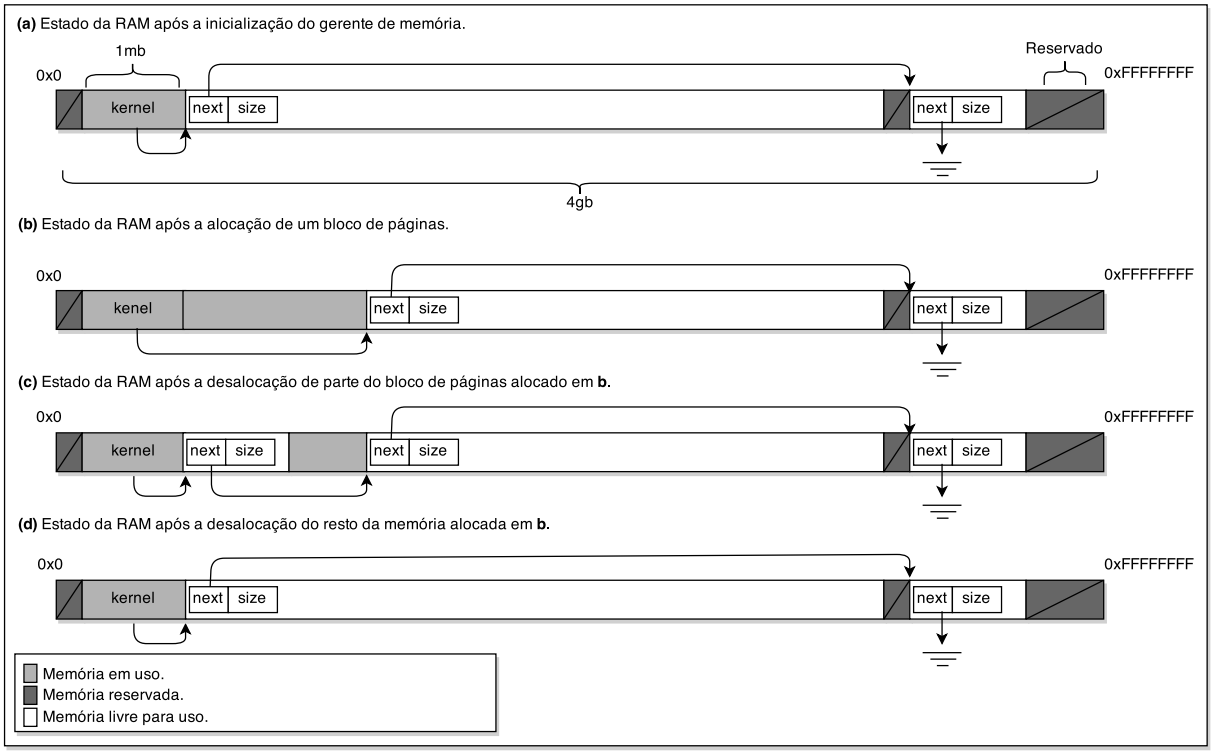


Figura 3.2: Gerenciamento de memória.

O número máximo de interações é dado pela Fórmula 3.1. Se analisarmos esta fórmula de forma minuciosa notaremos que na prática o número máximo de interações é muito baixo. Em uma máquina x86, com 4 gb de RAM e com páginas de 4 KB obteríamos um resultado de aproximadamente 500000 interações para o pior caso. Além disso, o pior caso só difere do melhor caso quando existe uma variação ínfima entre o tamanho das páginas. Para observar isso, basta imaginar que todas as páginas são do mesmo tamanho, isso faria a complexidade do pior caso cair para O(1). Assim, a fragmentação máxima seria algo muito difícil de acontecer em uma máquina virtual, pois na prática os programas apresentam tamanhos variados e não tão discrepantes.

$$n = \frac{TAMANHO\_MEMORIA - MEMORIA\_RESERVADA - MEMORIA\_EM\_USO}{2 * TAMANHO\_PAGINA} \tag{3.1}$$

Um outro ponto marcante, que merece ser observado a respeito desse algoritmo, é que o pior caso será insignificante para uma máquina virtual, pois o número de interações serão bem próximos de 1, já que é esperado que a VM aloque a maior parte da memória disponível no início de sua execução. Assim, a variável MEMORIA_EM_USO será muito próxima da variável TAMANHO_MEMORIA, forçando que o n seja ainda menor.

Como foi mencionado anteriormente, o algoritmo possui um custo de utilização de memória constante, que é o fator que acreditamos ser mais importante nesta situação. Já que quanto menor for a quantidade de memória gasta para gerenciar o sistema, mais memória sobrará para a máquina virtual e consequentemente para o sistema operacional *guest*. Como algo maior que uma interação para alocar e desalocar uma página será pouco frequente, acreditamos que esta seja uma solução apropriada para uma máquina virtual nativa.

Como descrito anteriormente, nesta primeira versão do algoritmo utilizamos a política *First Fit* de alocação. Em um sistema operacional de propósito geral isso poderia ocasionar uma alta fragmentação externa, em virtude da diferença de tamanho entre as regiões alocadas. Nesse primeiro momento, a política *First Fit* ordenada pelo endereço de memória parece adequada, já que a quantidade de processos em uma máquina virtual, teoricamente, deveria ser limitada e comportada. A depender dos requisitos da máquina virtual, uma política de alocação *Best Fit*, por exemplo, poderia ser utilizada. A escolha da melhor política, vai depender de como a máquina virtual será construída e quais serão os requisitos adicionais. Dessa forma, optamos pelo uso de uma política mais simples e facilmente modificável.

É possível aplicar algumas otimizações, tornando esse algoritmo mais eficiente. Algumas técnicas de gerenciamento de memória fazem uso de uma *hashtable*, indexados pelo tamanho dos bloco de páginas mais requisitados. Essas soluções são mais eficientes na detecção de áreas livres, pois dependendo da implementação, é possível reduzir a fragmentação externa e o número máximo de interações para encontrar uma página física disponível. Algumas otimizações interessante foram discutidas por Wilson e outros, como a introdução de um limite de tamanho na escolha do bloco que terá páginas removidas, reduzindo a fragmentação da lista de blocos livres [86].

Na seção sobre projetos futuros, discutiremos com mais detalhes o que pode ser investigado em futuras implementações de gerenciamento de memória. Apesar de ser susceptí-

vel a melhorias, acreditamos que o algoritmo aqui implementado se comportará de forma razoável, quando usado para gerenciar grandes blocos de memória da máquina virtual, conforme descrito anteriormente.

3.3 Gerenciamento de arquivos

Uma característica desejável em praticamente todo sistema operacional é a capacidade de manter e recuperar informações salvas por períodos longos de tempo. Isso possibilita que o sistema operacional seja construído de forma modularizada, adequando-se melhor às configurações de *hardware* e *software* de um determinado usuário. Contudo, o simples fato de manter os dados por longos períodos de tempo não é suficiente para a maioria das aplicações, pois não adianta o sistema operacional ser capaz de manter dados, se os programas não forem capazes de acessá-los. Dessa forma, é necessário um conjunto mínimo de chamadas de sistemas capazes de criar, ler e atualizar dados. Uma abstração muito comum para o acesso a dados em sistemas computacionais são os arquivos, que permite ao usuário o endereçamento e gerenciamento de informações de uma forma mais simples.

Um sistema de arquivos deve gerenciar a concorrência no acesso aos arquivos, sendo inaceitável perdas ou danos às informações do usuário. Essas exigências, somadas a detalhes de implementação, tornam o gerenciamento de arquivos uma das tarefas mais complexas de serem alcançadas, exigindo uma atenção especial no projeto e na sua implementação.

Apesar da complexidade, um gerenciamento de arquivos é algo extremamente desejável em um sistema operacional. Em alguns casos, quase todos os recursos de um sistema são interpretados como arquivos. No Linux, por exemplo, os dispositivos de entrada e saída são modelados dessa forma. Isso permite uma interação simples entre os programas e os *drivers*.

Sabendo disso, acreditamos que uma máquina virtual com um suporte mínimo para arquivos simplificaria consideravelmente o desenvolvimento. Na nossa infraestrutura, assim como no Linux, o sistema de arquivos é a base do funcionamento de diversos serviços do *kernel*. Todos os módulos carregados dinamicamente são armazenados como binários ELF. Isso simplifica o trabalho de atualização e testes dos módulos da máquina virtual, já que por sua vez, são compilados de forma separadas, permitindo atualizações em tempo de execução.

O módulo de sistemas de arquivo foi implementado seguindo os mesmos conceitos apresentados no gerenciamento de memória. Optamos por dividir o gerenciamento de arquivos em duas camadas, a camada física e a lógica. Essa divisão, incluindo o fluxo de comunicação entre as camadas, pode ser visualizada na Figura 3.3.

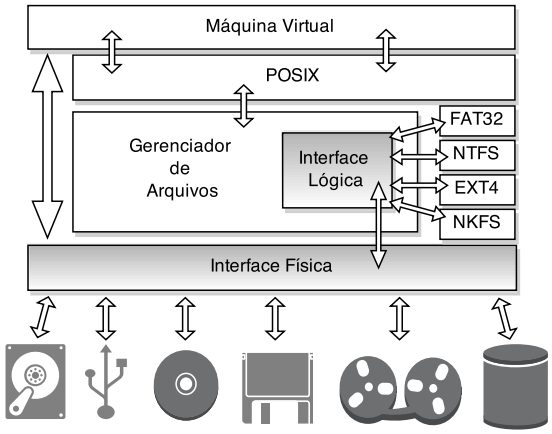


Figura 3.3: Gerenciamento de arquivos.

A camada física é responsável por abstrair o comportamento do dispositivo de armazenamento, incluindo o seu correto gerenciamento. Infelizmente, esses dispositivos possuem certas peculiaridades que devem ser respeitadas nas operações de leitura, escrita e busca. Assim, esta camada apresenta uma interface única e com um conjunto mínimo de rotinas, capazes de simplificar o acesso aos dispositivos de armazenamento instalados.

Já a camada de gerenciamento lógico, está ligada ao gerenciamento de arquivos. Essa interface é independente de implementação, sendo possível utilizar qualquer sistema de arquivos conhecido. Foi projetada seguindo as especificações do padrão POSIX, sendo obrigatório a implementação de todas as funções que operem sobre arquivos. Sobre esta camada, implementamos um sistema de arquivos mínimo, o NFSII. Este sistema de arquivos foi construído pensando no comportamento da máquina virtual e pode ser considerado uma fonte de referência para novas implementações. Nas próximas seções exploraremos com maior profundidade o funcionamento dessas camadas.

3.3.1 Camada física

Do ponto de vista prático, controlar e gerenciar dispositivos é um processo mecânico e tedioso. Cada dispositivo de armazenamento responde a um determinado protocolo de comunicação. Uma unidade pode exigir um tempo mínimo de espera antes do recebimento de um comando, aguardar uma interrupção ou até mesmo a liberação de uma *flag*.

Em uma unidade ATA[3], que é um padrão de interface para dispositivos de armazenamento, por exemplo. Para que seja possível ler um único setor de 512 *bytes* para memória, de forma direta, sem o auxílio de *drivers* ou do sistema operacional, é necessário executar

[3]Acrônimo para *Advanced Technology Attachment*.

ao menos uma dezena de operações correlacionadas [5, 25], tais como: verificar se o disco está presente, verificar se o disco está executando alguma operação, reiniciar o disco em caso de falha, enviar o comando de leitura, configurar o tamanho do bloco que será retornado, ativar o modo de operação (DMA[4] ou PIO[5]) e verificar se a controladora está pronta ou apresenta erros, em cada passo realizado. Como podemos notar, as operações descritas anteriormente são bastante repetitivas e podem, inclusive, exigir uma série de outras configurações prévias, tais como configurar o sistema de interrupção, alocar espaços em memória e controlar concorrência no acesso ao dispositivo.

Sabendo disso, claramente concluímos que seria necessário um nível mínimo de abstração dos dispositivos de armazenamento. A interface proposta é composta por diversas funções. As quatro principais são: seek, read, write e shift. Essas funções permitem que qualquer dispositivo, capaz de armazenar e ler informações, possa ser utilizado pelo módulo lógico[6]. Uma máquina virtual, também pode se beneficiar deste módulo, acessando o dispositivo de forma direta, sem a supervisão da camada lógica. Isso reduz consideravelmente o custo no acesso ao dispositivo, mas é necessário desativar a camada lógica deste dispositivo. Fazendo com que, o mesmo seja entregue completamente à máquina virtual.

Uma característica importante desta interface é o fato dela utilizar LBA como método de endereçamento. O LBA, ou *Logical Block Addressing*, é uma técnica de endereçamento criada para substituir o antigo modelo CHS[7]. No CHS, para efetuar uma leitura ou escrita, precisávamos escolher o cilindro, o setor e a cabeça das unidades magnéticas e óticas. O LBA é uma forma de endereçamento mais simples e eficiente, que permite acessar as unidades magnéticas através de um endereço lógico.

Escolhemos utilizar o LBA 64 em virtude do endereçamento em dispositivo de armazenamento serem capazes de ultrapassar o endereço máximo gerenciável por um registrador. Dessa forma, as camadas superiores enxergam a camada física como uma sequência de blocos lógicos, indexados por um inteiro de 64 *bits*.

Com interface física as camadas superiores enxergam o *hardware* de armazenamento como um vetor, acessado através de blocos de tamanho fixo. Quando efetuamos um seek, o endereço LBA passado é convertido para o endereço físico correspondente. Em computadores que não suportam LBA diretamente, esses valores podem ser convertidos para CHS. Em fita magnética, por exemplo, podemos fazer uma rotação relativa à posição inicial.

As leituras e escritas são feitas sob demanda, bem diferente da prática, onde só é possível salvar blocos fixos de dados em disco. A interface física possui uma *cache* pequena que armazena informações antes da operação *flush*. Dessa forma é extremamente simples

[4] Acrônimo para *Direct Memory Access*.
[5] Acrônimo para *Programmed Input Output*.
[6] Detalhes relacionados a implementação podem ser encontrados no Anexo I, na classe Storage.h
[7] Acrônimo para *Cylinder*, *Head* e *Sector*.

ler e salvar dados em uma unidade física, permitindo que as camadas superiores não se preocupem com o tratamento de interrupções ou com o gerenciamento do DMA. Além disso, inserimos uma operação de movimentação dentro da *cache*, o *shift*. Acreditamos que isso vai tornar o desenvolvimento ainda mais simples, pois com isso é possível navegar dentro do bloco lógico sem a necessidade de carregar o conteúdo da *cache* para memória.

Atualmente, a memória é o único dispositivo utilizado pela interface física, mas isso não representa uma limitação. No futuro poderemos criar *drivers* para trabalhar com disco, CDs, DVDs, unidades *flash* etc. O sistema de arquivo executado em memória foi uma escolha de projeto, para reduzir o trabalho de escrita de *drivers*. Assim, no momento, parte da memória é vista como um dispositivo de armazenamento secundário e todo o acesso de escrita de dados é enviado diretamente para uma imagem de disco armazenada em memória. Essa imagem é construída no momento da compilação e será descrita com mais detalhes nas próximas seções.

Na ideia original, quando estávamos projetando o mecanismo de gerenciamento de arquivos, a máquina virtual seria responsável por gerenciar os dispositivos físicos. Por esta razão, o sistema de arquivos deveria ser projetado inteiramente em memória, não necessitando de uma interface para armazenamento persistente. Contudo, com o passar do tempo, essa implementação se mostrou muito limitada, pois o *kernel* não seria capaz de manter informações por grandes período de tempo, sendo necessário detectar o *hardware* e se adaptar a cada *boot*. Além disso, não seria interessante gastar memória para armazenar informações utilizadas pouco frequentemente, como configurações de vídeo, endereços de dispositivos, mapa de memória etc. Isso poderia se tornar um problema em futuras atualizações do sistema. Dessa forma, optamos por projetar uma interface mais robusta, capaz de abstrair o *hardware* de armazenamento, deixando a cargo dos programadores da máquina virtual a escolha de como seria esse funcionamento: usar a nossa infraestrutura como um sistema operacional, fazendo uso do disco através das abstrações de diretórios e arquivos, ou de uma forma mais agressiva, acessando os dispositivos diretamente sem o auxílio do *kernel*.

Independente da forma como a máquina virtual for projetada, ela poderá se comunicar diretamente com a interface física, sem qualquer contra indicação, simplificando a utilização de dispositivos já detectados e controlados pelo *kernel*. Contudo, precisamos ficar atento ao problema de concorrência de acesso direto, onde um dispositivo em uso pelo *kernel* pode sofrer com interferências da máquina virtual.

Como durante a execução, a máquina virtual pode requerer acesso privilegiado a interface do dispositivo. Após a utilização, a máquina pode deixar o dispositivo em um estado inconsistente. Dessa forma, é importante ter em mente que a máquina virtual deve conhecer parte do funcionamento do núcleo, para evitar falhas catastróficas durante a execução. Caso a máquina virtual queira usar a UART diretamente, por exemplo, sem

nenhum auxílio do *kernel*, é recomendável que o módulo UART padrão seja desligado. O mesmo vale para o vídeo, teclado e qualquer outro dispositivo abstraído pelo *kernel*. Na Seção 3.5 discutiremos com mais detalhes a implementação dos drivers.

3.3.2 Camada lógica

Em sistemas operacionais de dispositivos móveis é comum os desenvolvedores optarem por esconder o sistema de arquivos. Isso simplifica a utilização do aparelho, já que o gerenciamento dos arquivos e dados gerados pelo usuário passam a ser de posse exclusiva da aplicação. Analogamente, a camada lógica opera da mesma maneira. Ela remove do programador a tarefa de gerenciar o sistema de arquivos diretamente, fornecendo uma interface conveniente para aplicação.

O programa executado sobre a nossa infraestrutura enxerga o dispositivo como um diretório. Nesse diretório o aplicativo é capaz de executar operações de leitura, escrita, busca e deleção de arquivos. Contudo, os programas não são capazes de ver os binários e bibliotecas do *kernel*. Isso torna o trabalho do programador extremamente confortável, já que ele não precisa se preocupar com acidentes, como deletar um arquivo importante ou alterar uma configuração do sistema. O programador pode simplesmente criar arquivos e pastas onde ele achar conveniente.

Dessa forma, teoricamente é possível utilizar qualquer sistema de arquivos existente. O projetista da máquina virtual pode optar por implementar um módulo FAT32, NTFS, EXT4 ou qualquer outro sistema de arquivos para substituir o sistema padrão. Um sistema de arquivos conhecido, tanto pelo sistema operacional hospedado como pelo *kernel*, pode permitir um compartilhamento de dados de forma mais simples e conveniente. Assim, acreditamos que este é um recurso desejável em sistema operacional que rode abaixo da máquina virtual.

Apesar do gerenciamento de arquivos ser bastante versátil, optamos por criar uma organização mínima de diretórios. Escolhemos adotar o mesmo padrão do *UNIX*, pelo fato de sua organização ser bem conhecida e simplificar a compatibilidade com o padrão *POSIX*. Na Tabela 3.3, apresentamos todas as pastas presentes no diretório raiz, incluindo o propósito de cada uma. Como comentamos anteriormente, isso é apenas uma recomendação. O programador da máquina virtual pode modificar esse diretório como achar melhor, contudo devemos ter em mente que a falta de padronização pode causar problemas no futuro, já que não é permitido dois arquivos possuírem o mesmo nome. Logo, é preciso ter cautela ao definir uma padronização diferente da sugerida.

Um detalhe importante, que merece ser enfatizado, é que na nossa infraestrutura qualquer programa inserido na pasta *boot* será carregado e executado automaticamente. Essa pasta é procurada logo após a inicialização do sistema de arquivos. Isso é uma

maneira de simplificar o processo de configuração do sistema, deixando o programador livre para escrever programas de *boot*. Com essa função é possível executar pequenos programas nativamente, bastando copiar os binários para pasta boot no momento da compilação.

Prefixo	**Descrição**
/	Referência ao diretório raiz.
..	Referência ao diretório pai.
.	Referência ao diretório atual.
~	Referência à pasta *home/vm*.
Pasta	**Descrição**
bin	Programas, comandos e utilitários relevantes.
dev	Dispositivos essenciais reconhecidos pelo *kernel*.
etc	Arquivos de configuração.
lib	Bibliotecas e *plugins* utilizados.
tmp	Arquivos temporários.
var	Variáveis mapeadas em arquivos.
boot	Programas carregados após a etapa de *boot*.
home/vm	Pasta destinada às máquinas virtuais.

Tabela 3.3: Organização do sistema de arquivos.

A camada lógica também possui algumas funções que não estão presentes na interface POSIX. Essas funções podem ser acessadas em modo privilegiado. Um dos motivos dessa divisão é que existe alguns detalhes que não estão implementados no padrão POSIX e podem beneficiar o funcionamento da VM, como por exemplo suporte para alocação contígua de dados.

Em máquinas virtuais, é comum o uso de alocação contígua da imagem de disco. Por exemplo, caso o usuário queira destinar 10 gb para o sistema hospedado, a VM pode criar múltiplos aquivos que representam esta imagem, ou pode optar pela criação de um bloco contínuo de memória. Os blocos contínuos, possuem a vantagem do acesso sequencial. Nesse tipo de imagem, um HD mecânico pode se beneficiar da localidade espacial das informações, não necessitando mover o cabeçote para posição remotas da imagem. Esse tipo de operação depende do sistema de arquivos, pois sem um suporte para alocação contínua o sistema passa a fragmentar os dados em disco, gerando implicitamente o mesmo resultado da criação de múltiplos arquivos. Além disso, a depender do sistema de arquivos, as imagens podem ser limitadas em tamanho. Por exemplo, no FAT32 o tamanho máximo de um arquivo é de 4 gb, logo seria impossível alocar um arquivo de 10 gb contínuos de

dados em disco. Contudo, em sistemas modernos, como EXT4 e NTFS, essas operações são permitidas.

A alocação contígua também permite criar partições virtuais dentro de um sistema de arquivos. A vantagem disso é a capacidade de mover os dados com maior facilidade quando, comparamos com as partições lógicas. Como o suporte para alocação continua é dependente do sistema de arquivos, a interface lógica tenta criar uma partição continua quando requisitado. Caso o sistema de arquivos não suporte essa operação, o *kernel* substitui a chamada para a alocação tradicional.

3.3.3 Sistema de arquivos

Basicamente, um sistema de arquivos é uma organização lógica dos dados armazenados em um dispositivo físico. Ou seja, é uma padronização da forma como os dados devem ser gravados e lidos em uma unidade de armazenamento. Conforme mencionando anteriormente, a camada lógica interage com a implementação do sistema de arquivos e com a camada física, logo a nossa infraestrutura é bastante versátil, não ficando restrita a uma única implementação.

Contudo, apesar da versatilidade, o sistema não é capaz de funcionar sem uma implementação mínima de um sistema de arquivos. Logo, analisando o comportamento das máquinas virtuais hospedadas, projetamos um sistema de arquivos extremamente simples, o NFSII. Esse sistema é baseado no NFSI, o sistema de arquivos padrão do Neutrino OS, não tendo nenhuma ligação com o NFS - Network File System.

Com base na análise dos principais sistemas de arquivos, chegamos à conclusão que as alternativas populares ao NFSI - como por exemplo a família EXT - eram extremamente complexas de serem construídas em sua totalidade. Além disso, não seria justificável o uso de um padrão simples e proprietário como o FAT, já que o objetivos era criar uma solução completamente livre. A adoção do NFSI simplificou o trabalho e possibilitou que focássemos na implementação da infraestrutura como um todo, ao invés de gastar um precioso tempo portando um sistema de arquivos complexo.

Assim, criamos um sistema de arquivos com base em algumas premissas que consideramos importantes no projeto de uma máquina virtual. Acreditamos que um sistema de arquivos apropriado deveria ser capaz de trabalhar tanto em memória como em disco, consumir pouco espaço de armazenamento, ser extremamente simples, estruturado em diretórios e deveria ser capaz de operar sobre grandes volumes de dados de forma contínua. Com base nessas características, projetamos o NFSII.

Utilizamos uma árvore onde todas as folhas são representadas por arquivos. Na Figura 3.4 podemos visualizar como os arquivos são organizados logicamente. O primeiro nó é a raiz, este nó contém informações relativas ao ao sistema de arquivos, incluindo um

ponteiro para o primeiro diretório.

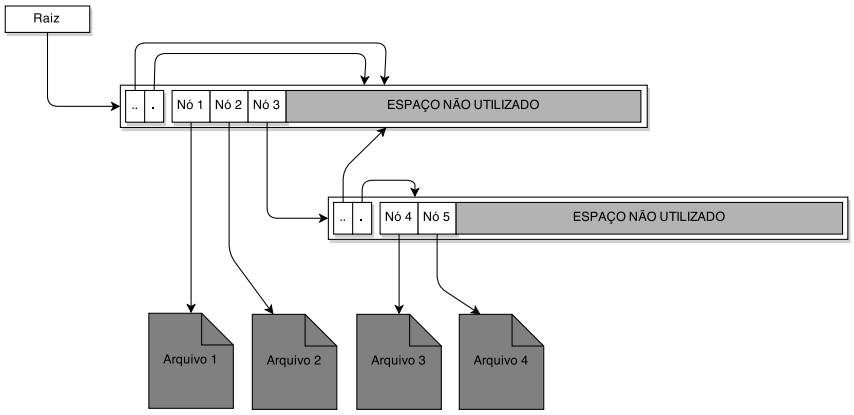


Figura 3.4: Árvore de diretórios.

Nesse sistema de arquivos, todos os nós possuem uma variável *type*, que determina o tipo do nó. Os nós do tipo pasta, são capazes de ter filhos. Qualquer outro tipo deverá obrigatoriamente ser uma folha. Ao todo, existem 255 tipos de nós, que podem ser usados livremente para diferenciar características dos arquivos.

Na implementação do NFSII optamos por usar um alocador similar ao apresentado na Seção 3.2.3. Sempre que é necessário criar um nó, o sistema requisita ao alocador uma quantidade de páginas em disco, suficiente para atender a necessidade da aplicação. Assim, caso seja necessário alocar, por exemplo, 100 gb, o alocador retornará um bloco sequencial de tamanho compatível e uma entrada na árvore diretórios será criada. Uma das vantagens dessa técnica é que os dados são sempre alocados de forma contínua, não ficando fragmentados em disco. Contudo, tem uma desvantagem clara: é necessário conhecer o tamanho do arquivo que será alocado previamente.

Para contornar este problema, a interface lógica foi projetada prevendo situações onde o sistema de arquivos necessita de uma expansão. No sistemas de arquivo que permitem expansão, incluindo o NFSII, quando um arquivo chega a um determinado tamanho é necessário alocar mais espaço em disco. Nas tabelas FAT, por exemplo, o arquivo passa a sofrer com fragmentação. No NFSII, simplesmente é feita uma nova alocação com tamanho compatível e todo o conteúdo é copiado de uma parte do disco para outra. Essa operação é extremamente custosa, mas acreditamos que não irá ocorrer com frequência. Já que, teoricamente, a maioria das operações da máquina virtual são de tamanho controlados.

Uma alternativa à técnica de expansão utilizada seria fragmentar o dado em disco, caso o arquivo necessite expandir. Isso pode ser feito facilmente, modificando a árvore do NFSII. Bastaria inserir uma tabela de endereços no lugar das folhas. Dentro dessa tabela

teríamos os ponteiros para os clusters em disco. Essa tarefa é extremamente simples de ser alcançada e pode ser um alternativa, caso a máquina virtual necessite expandir arquivos constantemente, algo que provavelmente não irá ocorrer com frequência.

Por ser um sistema extremamente simples, não implementamos o conceito de *Journaling*. Logo, o sistema é extremamente suscetível a falhas, não mantendo nenhum tipo de histórico de alterações. Além disso, não inserimos nenhuma estrutura de indexação para acelerar as buscas, algo que pode ser adicionado sem maiores problemas no futuro. Em virtude dessas características o sistema é extremamente leve e pode ser utilizado diretamente em memória.

O sistema foi projetado independentemente do dispositivo utilizado para armazenamento. Dessa forma introduzimos algumas constantes que podem ser modificadas, buscando uma melhor compatibilidade com a mídia utilizada. Como utilizamos a nossa implementação em memória, utilizamos páginas de 4 KB para simplificar o alinhamento com o gerenciamento de memória. Dessa forma é possível proteger áreas contra escrita, leitura ou execução. O tamanho da página pode ser configurado livremente. Dependendo da arquitetura alvo, o sistema de arquivos pode ser reconfigurado para obter um maior desempenho ou menor consumo de memória.

Existem alguns sistemas de arquivos bem avançados, que podem fazer parte de futuras implementações do módulo de gerenciamento de arquivos. A VMWARE, buscando otimizar o gerenciamento das imagens das máquinas virtuais em servidores, criou o VMFS. O VMFS é um sistema extremamente robusto, com suporte para *Journaling* e transações distribuídas.

Segundo Satyam B. Vaghani, tipicamente o número de arquivos de configuração variam de 30 a 100 por máquina virtual e esses arquivos geralmente são bem pequenos. Pensando nisso, o VMFS foi projetado para focar em grandes volumes de dados, como por exemplo as imagens das máquinas virtuais[84]. Na criação do NFSII pensamos de uma maneira similar. A maioria do armazenamento usado seria utilizado por imagens grandes, e operações que alteram o tamanho dessas imagens seriam pouco frequentes. Algo que justificaria o baixo desempenho na operação de expansão, por exemplo. No futuro, acreditamos que recursos similares ao apresentado pelo VMFS, ou até mesmo o próprio VMFS, poderão ser integrados a nosso infraestrutura.

3.4 Gerenciamento de processos

Segundo Tanenbaum, um processo é uma abstração de um programa em execução [80]. Em essência, podemos definir um processo como um conjunto formado por três componentes fundamentais, conhecidos como contexto de *hardware*, contexto de *software* e espaço de endereçamento.

O contexto de *software* é utilizado pelo sistema operacional para manter características relacionadas à utilização de recursos e estados dos programas em execução. Sempre que um arquivo é aberto, um programa requisita mais memória ou algum dispositivo periférico é utilizado, o sistema operacional atualiza o contexto de *software* associado ao programa que fez a requisição. Assim, quando o processo é finalizado o sistema operacional é capaz desalocar todos os recursos utilizados, restaurando o estado lógico do sistema. Além disso, o contexto de *software* mantém informações relevantes a respeito da execução de cada processo, como PID, prioridade de execução, tempo de vida e hora de criação.

O contexto de *hardware* funciona de forma similar ao contexto de *software*, mas mantém características físicas, como estados dos registradores, estado dos descritores, ponteiros para pilha e informações relacionadas ao *hardware*.

Por fim, temos o espaço de endereçamento, que é a área de memória onde o processo reside. Essa área pode ser dividida em diversas formas, principalmente em computadores com suporte para à virtualização do espaço de endereçamento. A divisão de espaço de endereçamento mais simples é a clássica, que divide a memória do processo em código, *heap* e pilha [55].

Como usamos uma arquitetura modular, parte do contexto de execução dos processos é compartilhado entre alguns módulos. Isso significa, que na prática, o contexto geral do processo não é armazenado completamente em um único registro.

O contexto de *hardware*, por exemplo, é compartilhado com a camada ARCH, como veremos com mais detalhes na Seção 3.7. Dessa forma, é responsabilidade do ARCH carregar e salvar o estado do sistema, sempre que um processo receber ou perder a posse do processador. Em virtude do contexto de *hardware* ser compartilhado com o ARCH, o escalonador não necessita gerenciar características dependentes de arquitetura, simplificando o processo de troca de contexto e a portabilidade.

Já no contexto de *software*, a maior parte das características dos processos são mantidas nos módulos de forma isolada, como por exemplo a lista de arquivos abertos é administrada pelo sistema de arquivos. Quando processo é finalizado, ao chamar a *syscall exit* o sistema desaloca todos os recursos utilizados pelo processo. Essa característica simplifica toda a infraestrutura e permite que os módulos possam ser executados de forma individual. Se o programador preferir, ele poderá utilizar somente o módulo de sistema de arquivos, por exemplo.

Em nossa arquitetura, o módulo de gerenciamento de processos é responsável por três tarefas fundamentais, que consistem em carregar os programas para memória, gerenciar a utilização do processador e salvar o contexto de execução dos processos. Nas próximas seções discutiremos com mais detalhes o módulo de gerenciamento de processos e quais características poderão beneficiar o desenvolvimento de novas máquinas virtuais.

3.4.1 O carregador

No modelo de computação moderno, os programas compilados são armazenados na forma de arquivos binários. Cada sistema operacional utiliza, por padrão, um formato específico.

No passado, antes da introdução do modo protegido em computadores pessoais, era comum a utilização de binários em formato *flat*[8]. O formato COM, por exemplo, utilizado no MS-DOS, era caracterizado por conter dados e instruções em uma mesma seção dentro do arquivo. Isso simplificava o trabalho de carregamento do binário para memória, pois como os dados eram alocados sequencialmente em uma única seção, os binários podiam ser copiados diretamente para memória sem qualquer alteração adicional.

Atualmente, o processo de carga é um pouco mais complexo. A introdução do modo protegido e a paginação possibilitou a divisão dos binários em seções. Além disso os programas atuais podem requerer a utilização de bibliotecas dinâmicas ou que o sistema inicie a execução em modo compatibilidade. Pensando nisso, diversos formatos surgiram para substituir os arquivos *flat*, que eram muito simples. Entre eles, podemos mencionar os dois mais utilizados, o ELF [16] e o PE [72].

O *Portable Executable Format*, ou PE, é o formato utilizado pela Microsoft em seu conjunto de ferramentas. Esse formato chegou a ser utilizado em diversos sistemas operacionais no passado, incluindo o BeOS e o ReactOS. Já o *Executable and Linkable Format*, ou ELF, é o formato mais difundido atualmente, sendo o formato padrão dos sistemas operacionais baseados no Unix, como Linux e FreeBSD.

Como neste projeto decidimos manter compatibilidade com o padrão POSIX e com o conjunto de ferramentas do Linux, optamos por utilizar o formato ELF em todos os nossos binários. Essa escolha trouxe diversos benefícios, entretanto agregou algumas dificuldades adicionais ao projeto. Como o formato ELF, por padrão, é utilizado tanto por bibliotecas, programas e módulos, antes de ser carregado para memória, é necessário que o carregador faça uma leitura das informações presentes nos cabeçalhos dos arquivos ELF, para definir as dependências do binário. Portanto, optamos por criar um módulo especializado nesta tarefa, o carregador.

O carregador é o modulo responsável por alocar a memória inicial de um processo, configurar a pilha, os endereços de páginas e carregar o binário do programa para memória. Esse módulo interage diretamente com o sistema de arquivos e as unidades de armazenamento, garantindo um nível adicional de abstração no processo de execução de programas. Além disso, o carregador configura as páginas em endereços específicos de memória e efetua o correto alinhamento dos binários, incluindo o preenchimento de endereços não inicializados com zeros.

[8]Termo utilizado para descrever arquivos binários organizados como um único bloco sequenciais, sem buracos e *headers*.

Até o momento, o nosso carregador não suportar arquivos binários com ligação dinâmica, somente binários compilados estaticamente. O principal motivo é que precisamos recompilar as bibliotecas utilizadas dinamicamente pelos programas de forma que elas sejam compatíveis com o nosso conjunto de chamadas de sistemas. Além disso, o carregador necessita posicionar essas bibliotecas apropriadamente, exigindo um gerenciamento adicional dos módulos. Dessa forma, optamos por simplificar o processo de carregamento de programas, deixando as funcionalidades adicionais para futuras atualizações.

É importante ressaltar que o carregador não faz parte do gerenciador de processos. Contudo, ele é uma peça muito importante no funcionamento do sistema. Na Figura 3.5 demonstramos uma visão geral da sequência de operações realizadas na criação de um processo. As arestas representam a ordem que as mensagens são transferidas entre os módulos.

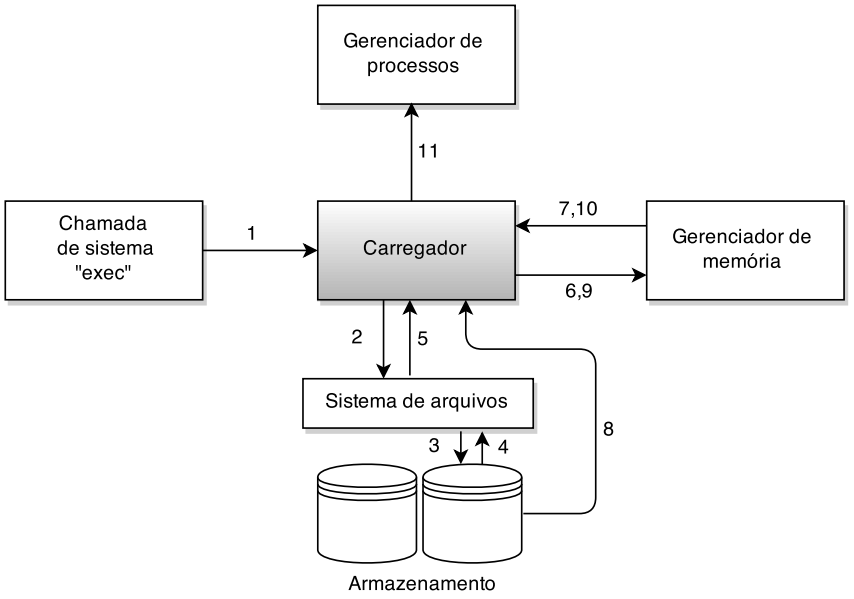


Figura 3.5: Funcionamento do carregador.

A inicialização de um processo começa na execução do método *exec*, da classe *system*. Esse método pode ser executado em modo usuário via uma chamada de sistema para a *syscall exec* ou diretamente via uma chamada procedural. Uma vez que o *exec* é invocado, ele envia para o carregador o endereço de um arquivo ELF que deverá ser executado. O carregador não conhece a organização dos dados em disco, muito menos qual o dispositivo de armazenamento que o binário reside. Desta forma, o carregador envia a cadeia de caracteres para o módulo de gerenciamento de arquivos, que é responsável por encontrar o arquivo dentro dos dispositivos de armazenamentos.

Uma vez que o arquivo é encontrado. O carregador varre o cabeçalho do ELF buscando informações relativas ao tamanho e alinhamento das regiões em endereços de memória. Essas informações são utilizadas pelo carregador para alocar e configurar a memória do processo. Nessa etapa, o carregador efetua as validações mais básicas, tal como se existe memória disponível e se o arquivo ELF é compatível com o processador e com os módulos do sistema.

Uma vez que as validações são completadas com êxito, o carregador solicita ao gerente de memória a alocação de um bloco contínuo de endereços com espaço suficiente para conter o processo e mais duas outras páginas. Uma das páginas extras será utilizada para armazenar o contexto de *hardware* e *software*, já a segunda, será utilizada exclusivamente para manter o diretório de páginas.

Após a alocação, o carregador carrega todo o binário do dispositivo de armazenamento para o espaço de endereçamento do processo e solicita ao gerente de memória a configuração das permissões. Atualmente, o sistema carrega o conteúdo do novo programa no tempo de vida do processo que solicitou a execução do *exec*. Contudo, no futuro esperamos utilizar suporte para *DMA* e *Bus Mastering*, que permitirá que o conteúdo do dispositivo de armazenamento seja copiado para memória enquanto o processador executa outras tarefas.

Por fim, o carregador inicializa todas as áreas de alinhamento com zeros, monta o contexto de *software* e passa o ponteiro da página que contém o contexto global do processo[9] para o escalonador.

3.4.2 O escalonador

O escalonador é o módulo responsável por gerenciar o tempo que cada processo permanecerá com a posse de um ou mais processadores. Isso é feito através de uma política de escalonamento. O foco da política pode variar dependendo do propósito da utilização do sistema. Os sistemas de processamento em lote, por exemplo, podem permitir a monopolização do processador até que uma tarefa seja concluída. Algo inaceitável em sistemas operacionais interativos, utilizados na maioria dos computadores pessoais, que devem ser projetados levando em consideração as interações assíncronas feitas pelo usuário. Essas diferenças na utilização reflete diretamente na concorrência por recursos e na forma como o escalonador é projetado.

Infelizmente, no modelo de computação atual, o processador é um recurso extremamente escasso. Em um computador pessoal, por exemplo, temos dezenas ou até centenas de processos disputando ativamente por uma única unidade de processamento. Em virtude

[9]No jargão de sistemas operacionais esse registro é conhecido como BCP - Bloco de controle de processos.

da grande concorrência por este recurso e a baixa disponibilidade, qualquer desempenho obtido na escolha do processo que será executado e na troca de contexto refletirá na quantidade de trabalho útil produzido por unidade de tempo.

Sabendo da importância de uma escolha adequada de uma política de escalonamento, optamos em nossa arquitetura dar o suporte necessário para implementações especializadas.

Grande parte da complexidade necessária para construção de um escalonador é abstraído por nossa infraestrutura. Isso permite que o programador tenha um maior controle da implementação da política de escalonamento. Dessa forma, dependendo dos requisitos da máquina virtual, o escalonador poderá ser modificado para melhor se adequar às características do sistema.

Por padrão, a nossa política de escalonamento é a *Round Robin*. *Round robin*, ou escalonamento circular, é uma política multitarefa extremamente simples de ser implementada. Esse algoritmo é utilizado em parceria com outras técnicas para criação de escalonadores multinível e já foi discutida exaustivamente ao longo da história [66, 75, 80]. O *Round-Robin* é uma política justa e que permite a preempção, que é um recurso extremamente conveniente, já que evita o desperdício de processamento quando um processo está esperando por periféricos ou dispositivos muito mais lentos que o processador.

Por questões de desempenho e simplificação, optamos pela utilização de uma lista duplamente encadeia circular com nó cabeça. Essa escolha, apesar de atípica, já que a maioria das implementações utiliza um escalonador FCFS com realimentação, onde sempre que o processo perde a posse do processador ele é inserido no final da fila, a nossa implementação possui baixa complexidade temporal na remoção e na troca de contexto.

Na nossa implementação não necessitamos esperar que o processo mude de estado ou varrer a lista dos processos prontos para que possamos remover uma entrada. Ou seja, conseguimos remover, inserir e, em alguns casos, buscar um processo em tempo constante, em O(1), mesmo que o processo seja um candidato para receber a posse do processador. Essa implementação foi utilizada pela primeira vez no Neutrino OS [63] e se mostrou bastante simples e relativamente eficiente.

Todo o algoritmo da política pôde ser implementado com menos de 100 linhas de código. Assim como no Neutrino OS, em nossa implementação, a lista de prontos é sempre mantida de forma constante em endereços fixos de memória. O processo de remover ou inserir uma entrada, consiste somente em alterar os ponteiros de próximo e anterior de dois nós da lista. Evitando as operações de remoção e inserção a cada término de um quantum[10].

Além disso, na nossa implementação o PID é o endereço de memória do registro que aponta para o BCP. O BCP, ou bloco de controle de processos, é um registro em memória

[10]Tempo que um processo gasta com a posse do processador.

que contém o contexto de *hardware* e de *software* de cada processo existente. O BCP é criado pelo carregador e é repassado para o escalonador. O PID, apesar de ser um endereço de memória, é um valor único, já que os registros dos processos devem estar sempre nas mesmas posições de memória para agilizar a troca de contexto. O registro é uma entrada que existe durante todo o tempo de vida do processo. Uma vez que o processo é finalizado o seu registro é desalocado.

Uma das vantagens de tratar o PID como endereço é a simplificação da validação do processo na operação de remoção, já que podemos acessar o processo na lista de prontos imediatamente - como uma tabela *hash* sem colisão. Além disso, o reaproveitamento de PIDs é automático, já que uma vez que um processo morre, a sua área de memória é liberada e poderá ser utilizada por um novo processo.

Na Figura 3.6 apresentamos uma visão geral da organização do algoritmo. A implementação é relativamente simples. Cada registro na lista aponta para uma página em memória que contém o contexto do processo. Quando o tempo de execução destinado ao processo acaba, a fila gira em sentindo horário, carregando o contexto de um novo processo.

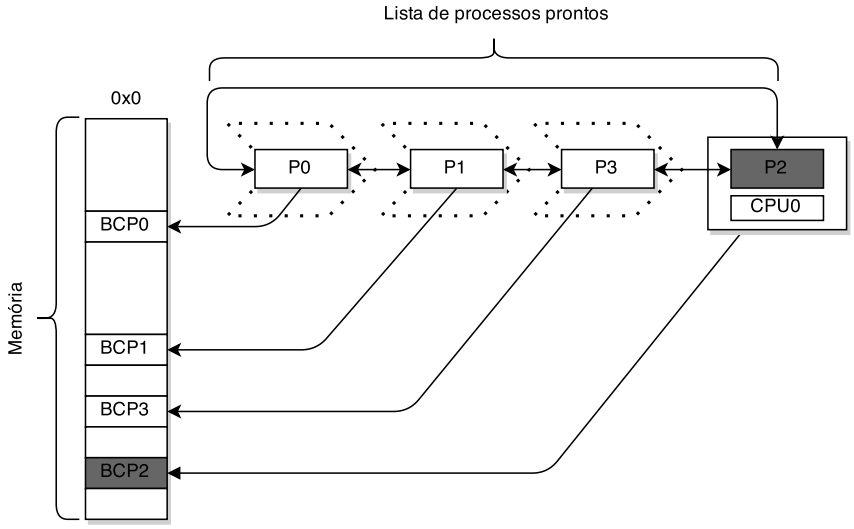


Figura 3.6: Política de escalonamento.

Apesar da nossa política de escalonamento padrão ter todos os recursos mínimos necessários para funcionar de forma multitarefa, por questões de simplificação, não inserimos um suporte adequado para controle de concorrência. Assim, até a versão atual da nossa infraestrutura, os processos são executados em lote, monopolizando o processador até que alguma interrupção seja acionada. Contudo, o processo para converter o sistema de monotarefa para multitarefa não será algo complexo, exigindo apenas a implementação

de semáforos e algoritmos *lock free*, que deverão prevenir condições de corrida em áreas específicas dos módulos, como por exemplo, no alocador de memória e no gerenciador de arquivos. Além disso, como a nossa infraestrutura é extremamente simples e modular, as áreas de concorrência são facilmente detectáveis. A grande maioria está na interface de comunicação entre os módulos.

Uma outra limitação é que atualmente o sistema trabalha de forma mono processada, subutilizando o recurso de processamento. Contudo, isso não é uma limitação da nossa infraestrutura e sim uma limitação no reconhecimento do ARCH. No futuro esperamos aprimorar o ARCH e utilizar toda a capacidade de processamento do sistema.

3.5 Gerenciamento de entrada e saída

Em virtude da elevada modularização dos computadores modernos, os sistemas operacionais atuais não interagem diretamente com os periféricos, e sim com interfaces responsáveis por enviar e/ou receber comandos das controladoras. Dessa forma, quando enviamos um dado para o monitor, por exemplo, o que estamos fazendo na verdade, é enviando uma série de comandos padronizados para uma interface[11]. Estes comandos, por sua vez, serão encaminhados para o *hardware* gráfico, onde serão decodificados e executados, gerando como resultado uma imagem no periférico de saída.

É importante ressaltar que, para que exista a comunicação entre o sistema operacional e um dispositivo, é necessário a utilização de um protocolo único na comunicação entre as duas partes. Portanto, caso um dispositivo implemente um protocolo diferente, o sistema operacional não utilizará o dispositivo corretamente.

Em virtude da grande variedade de protocolos e implementações de *hardware* - em alguns casos em virtude da ausências de padronizações mais rígidas no acesso aos dispositivos de entrada e/ou saída -, surgiu a necessidade de se criar uma camada de abstração entre o sistema operacional e o dispositivo físico. Esta camada é denominada *driver*. Os *drivers* são programas responsáveis por intermediar o acesso aos dispositivos, implementando um protocolo padrão de comunicação que deve ser conhecido pelo sistema operacional. Isso permite que o sistema operacional se comunique com um periférico via uma interface de gerenciamento comum a todas os dispositivos de uma mesma classe, ficando sob responsabilidade dos *drivers* traduzir as requisições do sistema operacional para um protocolo conhecido pelo dispositivo.

Sabendo da complexidade envolvida no acesso aos dispositivos de entrada e saída, concluímos que seria importante fornecer um gerenciamento mínimo de recursos para as máquinas virtuais nativas, tal como a utilização do teclado e vídeo para simplificar a inte-

[11] A interface pode ser uma rotina do bios, um endereço de memória, uma controladora de um barramento ou um *driver*

ração com o sistema. Assim, o módulo de gerenciamento de entrada e saída foi construído com o objetivo de gerenciar e abstrair o funcionamento dos periféricos, incluindo todos os *drivers* e serviços associados.

O uso de uma interface para abstrair o acesso aos dispositivos possibilita que programas nativos, compilados com a nossa infraestrutura, possam acessar alguns periféricos sem a necessidade de implementar *drivers* ou conhecer detalhes técnicos do *hardware*. Nas próximas seções discutiremos com mais detalhes, como o gerenciamento de entrada e saída foi estruturado e quais as vantagens de fornecer um protocolo padronizado para a máquina virtual.

3.5.1 Drivers

Como mencionado anteriormente, os programas executados nativamente interagem com os periféricos via uma interface de comunicação. A interação com um periférico pode variar desde do envio de alguns *bits* para uma determinada porta até o uso de protocolos complexos. Dessa forma, para simplificar o projeto de máquinas virtuais, propusemos a utilização de *drivers* para abstrair o acesso aos periféricos.

Na Figura 3.7 é apresentada a interface *Device*, que a base da implementação dos *drivers*. Todos os *drivers* em nossa arquitetura devem respeitar esta assinatura, caso contrário, o sistema não conseguirá configurar os dispositivos corretamente.

```
#ifndef DEVICE_H
#define DEVICE_H

#include <ilistener.h>

  class Device : public InterruptionListener {
  public:
    virtual void isr( const SSID ssid, Registers &registers );
    virtual bool isReady() = 0;
  };

#endif
```

Figura 3.7: A super classe Device.

O processo de instalação e configuração de um *driver* é uma tarefa simples, consistindo em registrar o tratador de interrupções do *driver* no sistema de interrupções[12].

Um detalhe importante sobre a nossa implementação, é que a configuração do dispositivo deve ser feita pelo próprio *driver*. Isso é necessário já que o *kernel* não tem

[12]O sistema de interrupções será discutido na Seção 3.6

conhecimento de nenhuma funcionalidade do periférico ou detalhes da arquitetura. Dessa forma, o protocolo usado na comunicação é extremamente importante, exigindo que um conjunto especifico de funções sejam implementadas pelos *drivers*.

Similar ao Linux, optamos por fazer com que todos os periféricos fossem associados a arquivos, simplificando a interação com os *drivers*. Sabendo disso, na nossa arquitetura os *drivers* foram agrupados em três classes diferentes: InputDevice, OutputDevice e InputAndOutputDevice. Essas três classes herdam da super classe Device e representam a barreira que separa o programa nativo do protocolo de comunicação utilizado pelos periféricos.

A classe InputDevice, pode ser visualizada na Figura 3.8. Esta classe representa o modelo que deve ser seguido por todos os *drivers* implementados para periféricos de entrada. Um detalhe desta classe é que ela foi projetada pensando em periféricos capazes de somente enviar dados para o sistema, tais como teclado, mouse, câmera digital e microfones.

```
#ifndef INPUT_H
#define INPUT_H

#include <device.h>

  class InputDevice : public virtual Device {
  public:
    virtual size_t read( char* buffer, size_t size );
    virtual void read( char &c ) = 0;
  private:

  };

#endif
```

Figura 3.8: A subclasse InputDevice.

Como podemos notar, a classe InputDevice exige a presença de uma função *read* em todos os *drivers* derivados desta interface. Essa característica, faz com que a interação com os *device* se resuma a uma chamada para uma função *read*. Por exemplo, caso o programador necessite ler um dado de um dispositivo de entrada, utilizando o nosso *driver*, tudo que ele necessitará fazer é chamar a função *read* do *driver*, independentemente se o dispositivo é um teclado, *UART* ou um dispositivo de armazenamento.

Seguindo a mesma linha de raciocínio, usada na elaboração dos *drivers* para dispositivos de entrada, temos a classe OutputDevice, que pode ser visualizada na Figura 3.9.

```
#ifndef OUTPUT_H
#define OUTPUT_H

#include <device.h>

  class OutputDevice : public virtual Device {
  public:
    template <typename T> void writeln( T value, uint8_t base );
    template <typename T> void writeln( T value );

    virtual void write( const char value ) = 0;
    virtual size_t write( const char* buffer, size_t size );
    virtual void reset();

    void write( const char* value );
    void write( const bool value );
    void write( const void* addr, uint8_t base = 10 );
    void write( uint32_t value, uint8_t base = 10 );
    void write( int32_t value, uint8_t base = 10 );
    void write( int64_t value, uint8_t base = 10 );
    void write( uint64_t value, uint8_t base = 10 );
    void writeln();
  private:

  };

#endif
```

Figura 3.9: A subclasse OutputDevice.

A classe OutputDevice deve ser implementada por *drivers* destinados a dispositivos de somente saída, tais como vídeo, impressoras, alto-falantes etc. Um detalhe desta classe, é que por conveniência, inserimos algumas rotinas para simplificar o processo de impressão em modo *kernel*. Desta forma, um *device* de saída contém algumas rotinas utilitárias associadas, como suporte para quebra de linha e impressão de tipos primitivos, que na prática não são utilizadas em modo usuário.

As funções extras, inseridas nos *driver* derivados de OutputDevice, permitem uma formatação rústica dos *logs* de saída. Além disso, no momento da inicialização do *kernel*, os módulos podem enviar mensagens para os *drivers* via as rotinas utilitárias, gerando *logs* mais simples de serem compreendidos. Como por exemplo imprimir quais *bits* de um determinado registrador estão ativados no momento da inicialização do sistema.

Como os *drivers* são independentes do sistema, um programa nativo compilado com algum dos nossos *drivers* poderá acessar as funções utilitárias, sem a necessidade de utilizar qualquer outra biblioteca.

Além dos *devices* somente entrada e somente saída, alguns periféricos respondem de forma bidirecional, recebendo e enviando informações. Dessa forma, para evitar que *drivers* bidirecionais sempre sofressem com um *downcast*[13] para os tipos InputDevice ou

[13]Termo utilizado para descrever a conversão de um objeto pai em um objeto filho na hierarquia

OutputDevice, criamos uma classe específica para esse tipo de periférico, que pode ser visualizada na Figura 3.10.

```
#ifndef IODEVICE_H
#define IODEVICE_H

#include <inputdevice.h>
#include <outputdevice.h>

   class IODevice : public InputDevice, public OutputDevice {
   public:

   private:

   };
#endif
```

Figura 3.10: A subclasse InputOutputDevice.

O InputOutputDevice deve ser implementado por dispositivos que podem tanto enviar como receber informações, tais como disco rígido, placas de rede, *UART*, *BIOS* etc. A implementação é extremamente simples e consiste, até o presente momento, de uma herança múltipla.

Implementamos ao todo cinco *drivers* diferentes: um *driver* para UART, capaz de enviar e receber informações pela porta serial; um *driver* para vídeo em modo texto, compatível com praticamente todos computadores atuais; um *drive* VESA, compatível com a maioria das placas de vídeo fabricadas a partir de 1995; um *driver* para teclado, compatível com os padrões USB e PS2; e por fim, um *driver* virtual, que simula uma unidade de armazenamento.

Esperamos que os *drivers* implementados sejam compatíveis com a maioria dos computadores modernos. Em nossos testes, formos capazes de executar a nossa infraestrutura em diversos computadores. Assim, acreditamos que os nossos *drivers*, apesar de simples, poderão simplificar o desenvolvimento de novas máquinas virtuais. A utilização da nossa infraestrutura evitará a necessidade de se projetar *drivers* básicos, que não influenciam diretamente no desempenho da maioria das aplicações.

3.6 Chamadas de sistema

Em um sistema de computação, as informações geradas pelos múltiplos componentes trafegam por estruturas denominadas interfaces. A interface gráfica, por exemplo, apresenta

orientada à objeto.

uma série de botões, caixas de texto e diálogos que podem ser controlados através de um dispositivo de entrada. Ao clicar em botão, enviamos um sinal para que o programa em execução possa efetuar a operação requisitada. É interessante ressaltar que, para a maioria dos usuários, não é importante saber o que o sistema operacional faz no momento que o botão é clicado, apenas se o resultado gerado está correto ou não.

As interfaces são uma forma conveniente de abstrair as camadas inferiores, permitindo uma maior independência entre os múltiplos componentes de um sistema. O uso de interfaces padronizadas permitem uma maior portabilidade e, consequentemente, uma maior modularização dos múltiplos componentes de um sistema.

Analogamente à interface gráfica, que é responsável por abstrair a utilização dos programas, as *System Calls*, ou chamadas de sistema, apresentam uma interface entre os programas e o *kernel*. Basicamente, as *system calls* são compostas por um conjunto de rotinas padronizadas, que são capazes de solicitar ou executar serviços presentes no *kernel*. Por exemplo, para imprimir um caractere no dispositivo de saída padrão, o programa deve executar uma *Syscall Write*, que por sua vez, solicita ao *kernel* a impressão de um valor numérico no dispositivo correspondente. Sem essa interface, o programa não teria a capacidade de interagir com *kernel*, tornando o sistema praticamente inutilizável.

Praticamente todas as interações entre o sistema operacional e os programas são realizadas através das chamadas de sistemas. Assim, programas compilados para interfaces diferentes, são, de maneira geral, incompatíveis.

Na tentativa de obter compatibilidade de *software*, em 1988 foi criado o padrão POSIX. O POSIX é um acrônimo para *Portable Operating System Interface*. Consiste em um conjunto bem extenso de normas e assinaturas de funções, que todo sistema operacional deve possuir para que exista uma retrocompatibilidade entre os programas[33]. A compatibilidade completa com o padrão POSIX pode exigir um esforço muito grande, algo inviável para a maioria dos projetos. Contudo, na prática, apenas um conjunto bem reduzido de funções precisam ser implementadas para que a maioria dos programas possam ser executados.

Em sistemas incompatíveis com o POSIX, podemos emular o comportamento de interfaces diferentes, para possibilitar a execução de programas escritos para outros sistemas. O WINE [31], por exemplo, utiliza dessa abordagem. O WINE é uma ferramenta que foi construída para possibilitar a execução de programas do Windows em ambientes compatíveis com o padrão POSIX, utilizado principalmente no Linux, MacOS e FreeBSD. Essa ferramenta pode ser vista como uma camada adicional de *software* entre as aplicações e o sistema operacional. Sempre que um programa faz a requisição para uma função da API do Windows, o WINE remapeia a chamada de sistema para uma rotina equivalente do POSIX, possibilitando a execução do aplicativo com uma baixa perda desempenho.

No caminho inverso ao proposto pelo WINE, existe o Cygwin[9], que permite a execu-

ção de aplicativos projetados para Linux em ambientes Windows. O Cygwin possui um conjunto de bibliotecas dinâmicas que implementam as principais chamadas de sistemas do Linux. Os aplicativos de Linux devem ser recompilados com essas bibliotecas, para serem capazes de rodar no Windows. Logo, no Cygwin, as chamadas de sistemas não são simuladas, e sim, substituídas.

3.6.1 Interface POSIX

Uma das vantagens de oferecer um suporte POSIX nativo é que todos os módulos da máquina virtual, que não necessitarem de acesso privilegiado, poderão ser construído e testados em ambientes compatíveis com o POSIX. Isso traz como principal vantagem um melhor suporte para depuração e detecção de erros. Além disso, a adoção do POSIX minimiza a necessidade de rescrever ou portar bibliotecas básicas, possibilitando um maior reaproveitamento de código.

Infelizmente, em virtude do grande número de exigências impostas pelo padrão POSIX, não fomos capazes de implementar todas as chamadas de sistema para a versão atual do *kernel*. Dessa forma, a nossa infraestrutura apresenta um suporte reduzido de chamadas sistemas, mas já permite a execução de diversas aplicações compiladas para outros sistemas POSIX. Atualmente, apenas um conjunto limitado de programas que seguem o padrão POSIX podem ser portados para o nosso *kernel*. Conseguimos em nossos testes portar pequenos emuladores e programas simples sem muito esforço. Contudo, para portar grandes aplicações, no estágio atual, pode ser necessário expandir o sistema através da implementação de outras chamadas de sistemas.

As nossas chamadas de sistemas seguem a mesma padronização do Linux. Todos os registradores são passados na mesma ordem, e as interrupções foram numeradas e projetadas para se comportar da mesma maneira. Assim, no estágio atual de desenvolvimento, qualquer programa em modo usuário compilado com a nossa infraestrutura, que siga o padrão POSIX, poderá ser executado diretamente no Linux. Infelizmente, o inverso não é possível, em virtude da limitada compatibilidade com o POSIX e do número reduzido de funções implementadas.

Na prática, para se obter compatibilidade completa de chamadas de sistemas com Linux, seria necessário ao menos implementar algumas centenas de funções. O valor exato é bastante volátil e depende da versão do kernel e da arquitetura alvo. No i386, até a data de elaboração deste trabalho, eram mais de 383 funções, incluindo as depreciadas[17]. Dessa forma, seria necessário um esforço muito grande de desenvolvimento para chegar a um estágio de compatibilidade completa.

Como podemos imaginar, nem todas as chamadas de sistemas do Linux são requeridas por todas as aplicações. Assim, ao invés de implementar todo o conjunto de chamadas de

sistemas do Linux, optamos por implementar somente as chamadas de sistemas do padrão POSIX que eram realmente fundamentais para o desenvolvimento de aplicações. Nesse sentido, optamos por portar a NewLib, uma biblioteca que implementa grande parte das funções presentes na biblioteca padrão do C, mas utilizando apenas um conjunto reduzido de chamadas de sistemas - maximizando o total de aplicações que podem ser portadas para a nossa infraestrutura. Ao todo são 19 *System Calls* que um sistema operacional necessita implementar para se obter um suporte mínimo para programas compatíveis com POSIX. Em virtude do curto espaço de tempo, implementamos todas as funções requeridas pela NewLib, exceto a *System Call Fork*. Dessa forma, no estágio atual de desenvolvimento, não existe suporte para aplicativos *multithreads*. A a lista completa das funções requeridas pela NewLib pode ser encontrada na documentação oficial[20].

3.6.2 Implementação

Uma chamada de sistema pode ser implementada de diversas maneiras. A escolha depende do suporte oferecido pelo processador e dos requisitos do sistema operacional. Contudo, a maneira mais utilizada consiste em fazer uso das interrupções de *software*, trocando o contexto de execução sempre que uma chamada de sistema é realizada.

Como a configuração do sistema de interrupções varia de processador para processador, o sistema de tratamento de interrupções foi inserido na camada *ARCH*. Assim, sempre que o sistema é inicializado, a camada *ARCH* configura o sistema de interrupções, mapeado todas as interrupções de *hardware* e *software* para um mesmo procedimento, o *interrupt_handler*.

O *interrupt_handler* é um procedimento escrito em C, que pode ser encontrado no arquivo core.cpp. Este procedimento recebe como parâmetro o código da interrupção e um ponteiro para um endereço de memória que contêm o contexto dos registradores no exato momento da chamada.

Sempre que uma interrupção ocorre, a camada *ARCH* intercepta a interrupção e faz uma chamada para o procedimento *interrupt_handler*. É responsabilidade do *interrupt_handler*, decodificar o código recebido e executar uma chamada para o módulo correto. Isso é feito com uso de uma tabela que mapeia os *ids* das interrupções para os módulos do *kernel*. A decodificação é executada sempre em O(1) e o custo do procedimento é extremamente baixo.

Uma vez detectado que o código recebido pelo *interrupt_handler* é uma chamada de sistema, o estado dos registradores é analisado, buscando descobrir qual serviço deve ser executado. Buscando uma maior portabilidade, os registradores são organizados em uma ordem específica.

Independente da arquitetura, o primeiro registrador, presente no registro de estado,

sempre contém o número da chamada de sistema. Essa característica segue a mesma convenção de chamada de sistemas do Linux x86/AMD64, onde o número da chamada é sempre inserido no registrador EAX e os parâmetros nos registradores seguintes[68].

O registro de estado pode ser organizado estaticamente de formas diferentes, para se adaptar a outras ABIs. No Linux para ARM, o número da chamada de sistema é sempre inserido no registrador R7. Dessa forma, o registro de estado pode ser modificado para que a posição R7 passe a ser a primeira entrada do registro. Isso melhora a compatibilidade com ABIs diferentes e reduz o esforço ao portar o *kernel* para outras arquiteturas. Além disso, o uso desta padronização melhora a compatibilidade binária com o Linux.

Após decodificar o número da chamada de sistema, é feito um salto para o procedimento responsável por executar a ação requisitada. Todos esses passos são executados em modo *kernel*. A camada *ARCH* é responsável por trocar o contexto de execução antes e depois de cada chamada. Dessa forma, o *kernel* é sempre executado em modo protegido. Já as aplicações que fazem uso de chamadas de sistemas, são sempre executadas em modo usuário.

Buscando uma maior organização, reservamos duas linhas de interrupções para chamadas de sistemas. A linha[14] 0x30 e 0x80. A linha 0x80 segue a mesma padronização do Linux, ou seja, não pode ser modificada. Foi construída com o propósito de efetuar compatibilidade com aplicativos em modo usuário. Já a linha 0x30, foi projetada para ser exclusiva da máquina virtual, podendo ser expandida conforme a necessidade de cada projeto.

A vantagem de possuir uma linha exclusiva para máquina virtual é a especialização das operações. Um módulo da máquina virtual, escrito em modo *kernel*, terá uma comunicação exclusiva com as camadas em modo usuário da VM. Supondo que a máquina virtual necessite alterar alguma característica do comportamento do *kernel*, isso poderá ser feito por meio desta linha, implementando uma chamada de sistema com a funcionalidade requerida.

3.7 Camada ARCH

Na tentativa de elevar a portabilidade da nossa infraestrutura, criamos uma camada de *software* que abstrai funcionalidades estritamente dependentes de arquitetura. Esta camada foi nomeada ARCH. O ARCH é o primeiro módulo a ser executado após o *loader* e possui diversas atribuições. Entre as tarefas atribuídas ao ARCH, podemos destacar: suporte para execução de linguagens de alto nível, abstração do subsistema de memória, interação com o *boot loader*, configuração do processador e interceptação de interrupções

[14] Ou código identificador de chamada de sistema.

e exceções.

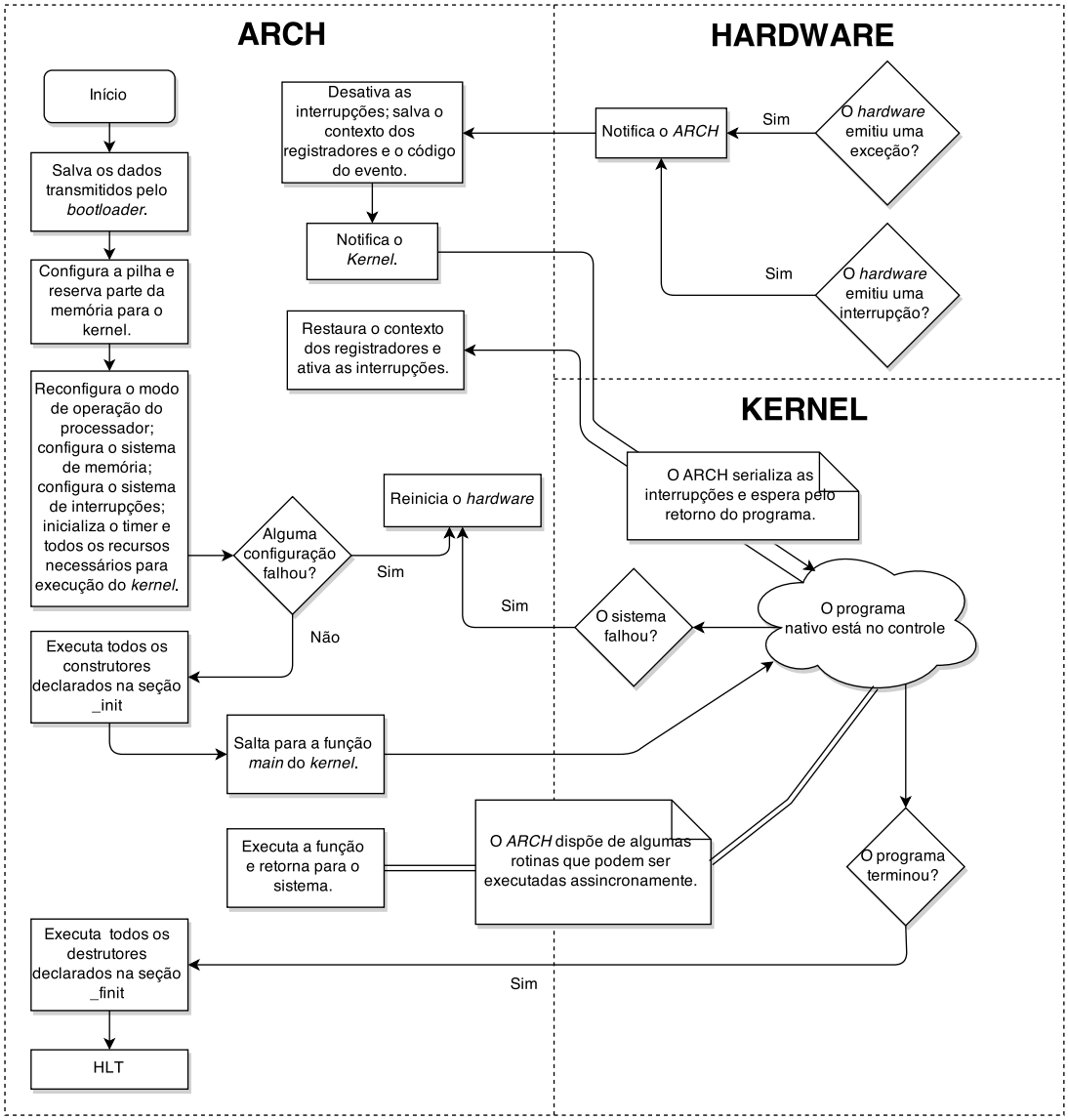


Figura 3.11: Fluxograma de execução da camada ARCH.

Diferente de todos os outros módulos mencionados neste capítulo, onde o uso era uma escolha do programador, todas as máquinas virtuais ou programas nativos, que quiserem se beneficiar dos recursos propiciados por esta arquitetura, devem obrigatoriamente ser compilados e ligados estaticamente ao módulo ARCH. A ligação estática é necessária por diversos motivos, entre eles, podemos enfatizar que o ARCH implementar a interface *multiboot*, tornando os aplicativos nativos compatíveis com diversos *boot loaders*, e consequentemente, viabilizando o carregamento dos binários em memória após a etapa

de *boot*. Além disso, o ARCH implementa o suporte mínimo requerido na utilização de linguagens de alto nível modernas, como C++. Dessa forma, todas as camadas superiores, construídas sob o ARCH, são extremamente dependentes da implementação desta camada.

Na Figura 3.11 é aprestando o fluxograma da execução do sistema, após a etapa de inicialização. No fluxograma descrevemos detalhadamente as operações realizadas pelo ARCH, incluindo o relacionamento com o *kernel*.

Como podemos notar na Figura 3.11, durante a etapa de inicialização, o ARCH é responsável por configurar a pilha e aloca memória estática[15] para o sistema. Além disso, todas as operações essenciais de configuração de *hardware*, que não são feitas pelo *bootloader*, são configuradas nesta etapa, como por exemplo, a inicialização do *timer*, do sistema de interrupções e configuração do sistema de paginação.

O ARCH também funciona como uma camada de abstração para programas nativos, executando todos os construtores declarados e saltando para o ponto de entrada do programa.

Como os programas compilados com ARCH são executados de forma nativa, temos que fornecer implementações alternativas de alguns binários utilizados na ligação de programas hospedados, com ênfase nos arquivos que definem os símbolos de inicialização[16]. Isso ocorre porque parte do processo de compilação é um acordo feito entre o sistema operacional, arquitetura alvo e o programa compilado. Assim, como os programas nativos não são executados sobre um sistema operacional, o ARCH é obrigado a redefinir os símbolos ausentes.

3.7.1 Interface nativa

Inserido no ARCH existem algumas implementações de rotinas que interagem diretamente com o *hardware*. Essas rotinas foram implementadas completamente em linguagem de montagem e são acessadas através de uma interface em C, nomeada interface nativa. As funções da interface nativa podem ser executadas a qualquer momento, após a transferência de controle para o aplicativo nativo.

Na Figura 3.12 temos as assinaturas de todas as funções implementadas até o momento. Essas funções são fundamentais para o funcionamento do *kernel*, e são utilizadas exclusivamente pelos módulos de gerenciamento de memória e processos. A interface completa, incluindo os endereços compartilhados de memória e registros podem ser encontrados no arquivo `arch.h`.

[15] A memória estática é utilizada principalmente para alocar tabelas de descritores que precisão ser mapeadas em regiões especificas de memória.

[16] No GCC/G++ esse símbolos são implementados nos arquivos crt0.o, crti.o, crtbegin.o, crtend.o e crtn.o.

```
extern "C" void _reboot();
extern "C" void _sstate( void* buffer );
extern "C" void _lstate( void* buffer );
extern "C" bool _pdupdate( void* pDir, uintptr_t vAddr, uintptr_t pAddr,
    uint8_t permission, size_t size = 1 );
extern "C" void _pdprotect( void* pDir, void* vAddr, uint8_t permission,
    size_t size = 1 );
extern "C" void _pdcreate( void* vAddr );
extern "C" void _pdload( void* pDir );
extern "C" void _pdsave( void* &pDir );
extern "C" void _ptcreate( void* pTable, void* pDir, uintptr_t base );
```

Figura 3.12: Assinatura das funções fornecidas pelo ARCH.

A vantagem desta interface é que ela abstrai o funcionamento do sistema de paginação. Desta forma, o trabalho de criar, atualizar, carregar e salvar tabelas de páginas, foram completamente abstraídos pelo ARCH, tornando os módulos de gerenciamento de memória e processos independente de plataforma. Além disso, o trabalho de salvar e carregar o estado do processador também é implementado pelo ARCH, simplificando a troca de contexto.

A interface nativa também atual na transferência de informações entre o ARCH e o *kernel*. Quando uma interrupção ocorre, por exemplo, o ARCH intercepta a rotina e transfere o fluxo de controle para uma função da interface nativa, a *interrupt_handler*. Essas ações são executadas em série para evitar o estouro de pilha, logo somente uma função pode ser executada por vez.

3.8 A organização das camadas

Como bem sabemos, uma falha em modo privilegiado pode provocar a perda de todas as informações geradas durante o tempo de vida de um processo. Por este motivo, sistemas operacionais modernos dividem o privilégio de acesso em camadas. As camadas mais próximas do usuário são aquelas que possuem o menor privilégio, enquanto as mais distantes são as mais prioritárias.

Nos sistemas operacionais modernos o privilégio é descrito por anéis. O anel 0 é o mais privilegiado. Um programa com este privilégio, pode acessar toda a memória disponível diretamente. Além disso, os conjuntos de instruções privilegiadas passam a ser acessíveis, permitindo que recursos de *hardware* possam ser reconfigurados. Já o último dos anéis, é o menos privilegiado. Qualquer tentativa de exercer uma ação de maior privilégio que a suportada resulta em uma exceção, e no pior caso, a morte do processo.

A organização do sistema é extremamente importante, pois, como sabemos, nenhum *software* é livre de falhas. Erros podem acontecer a qualquer momento, desta forma,

quanto menos programas executando em modo privilegiado, maior a segurança do sistema. Em contra partida, a troca de nível de privilégio exige que certas validações sejam efetuadas para garantir que um *software* malicioso ou um erro não prejudique a execução do *kernel*. Assim, quanto maior a quantidade de programas em modo não privilegiado, maior a perda de desempenho com trocas de mensagem e sincronização de informações. Portanto, no projeto de uma máquina virtual ou de um sistema operacional é necessário um balanceamento entre segurança e desempenho.

Como mencionado na Seção 3.1, a nossa infraestrutura pode apresentar variações, dependendo do privilégio requerido pela máquina virtual. A escolha do nível de execução é uma decisão de projeto, que pode variar de acordo com os requisitos de segurança e usabilidade dos módulos envolvidos.

Na nossa arquitetura existe basicamente dois anéis de proteção, o anel 0 e o anel 3. Contudo, deixa a cargo do programador a liberdade de escolher como o programa se comportará. Isso significa que no nosso *kernel*, um programa escrito pode ser executado em um nível privilegiado, em uma faixa de endereço posterior ao *kernel*, ou como um programa em modo usuário, sendo escalonado em função do tempo.

A liberdade na escolha do nível de privilégio só é possível graças ao funcionamento do *kernel*. Como a nossa infraestrutura se comporta como uma biblioteca, uma vez finalizadas as configurações do *kernel*, somente o escalonador é executado, em unidades de tempo regulares de 8.2ms. Assim, uma máquina virtual pode fazer uso do *kernel* de 3 maneiras diferentes, variando os níveis de privilégio de execução. Nas próximas seções discutiremos as três arquiteturas possíveis para uma máquina virtual.

3.8.1 Sistema em modo supervisor

Na organização em modo supervisor a máquina virtual é construída diretamente sobre a camada ARCH. Isso permite que a máquina virtual possa ser executada em modo privilegiado, tendo acesso direto a todos os recursos de *hardware* disponível.

Na Figura 3.13 podemos visualizar o relacionamento entre os módulos da arquitetura em modo supervisor. Como podemos notar, a máquina virtual passa a incorporar todos os recursos fornecidos pelo *kernel*.

Nesta arquitetura a máquina virtual tem a capacidade de acessar as chamadas de sistemas e os métodos públicos de todos os gerentes, diretamente, sem a necessidade de executar uma tradução de endereços ou validações de segurança. Isso quer dizer que caso a máquina virtual necessite usar alguma função presente no *kernel*, ela poderá fazer diretamente, sem a necessidade de recorrer a chamadas de sistemas. Seguindo essa lógica, na criação de um arquivo, por exemplo, poderia ser feita com a execução de uma única função do *filesystem*, a *create*, evitando todo o encadeamento de chamadas provocado pela

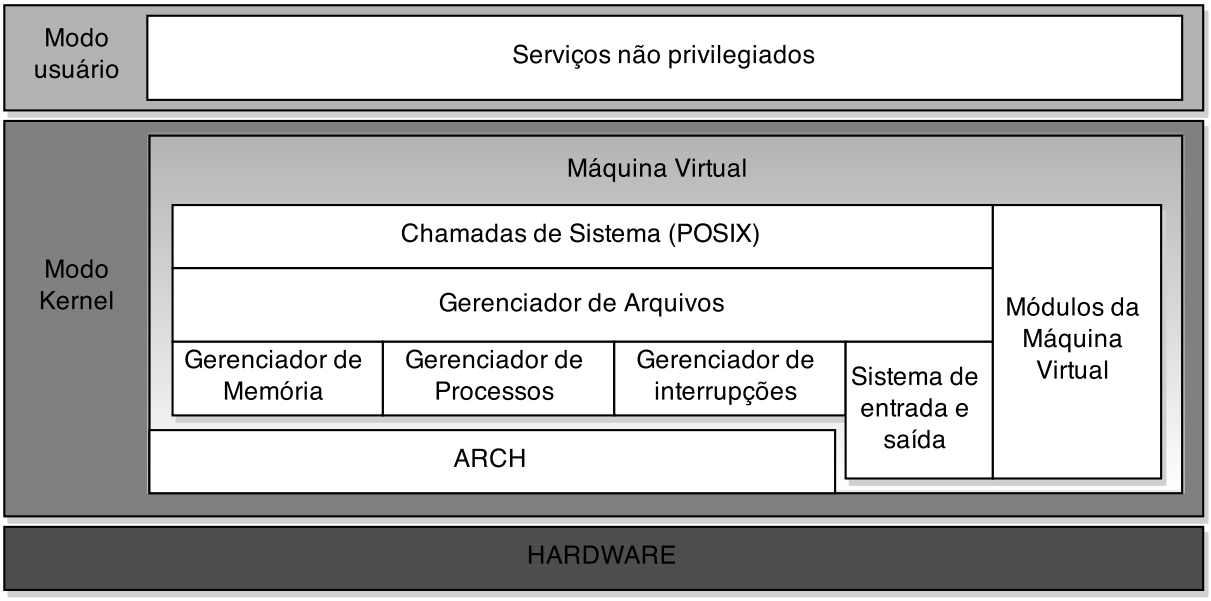


Figura 3.13: Organização em modo supervisor.

inclusão da NewLib.

Em virtude da infraestrutura ter sido projetada de forma orientada a objeto, somos capazes de reaproveitar e fornecer novas implementações, através dos recursos de herança e polimorfismo presentes no C++. Para fazer uso desta arquitetura, de forma efetiva, a máquina virtual necessita herdar da classe *System* e fornecer sua própria implementação. Assim, a máquina virtual é construída juntamente com o *hypervisor*, monopolizando todos os recursos do sistema.

A principal vantagem desta arquitetura, é o desempenho. Por estar em nível privilegiado a máquina virtual tem total controle sobre o sistema, podendo acessar qualquer módulo ou recurso diretamente. Contudo, existem algumas desvantagens. A máquina virtual passa a ser incompatível com o Linux, não podendo ser executada em modo usuário e ficando restrita ao nosso conjunto de ferramentas. Além disso, quanto maior a quantidade de código em nível privilegiado, menor a segurança, tornando essa implementação mais susceptível a erros de sistema.

3.8.2 Sistema em modo usuário

O sistema em modo usuário é o oposto da arquitetura em modo supervisor. Nesse modelo a máquina virtual é executada como um programa de sistema. Todo o acesso a *hardware* passa a ser intermediado pelo *kernel* padrão. Essa arquitetura, permite que múltiplas máquinas virtuais possam ser executadas paralelamente, já que o *kernel* escalona o pro-

cessador, memória e o acesso ao dispositivo de armazenamento.

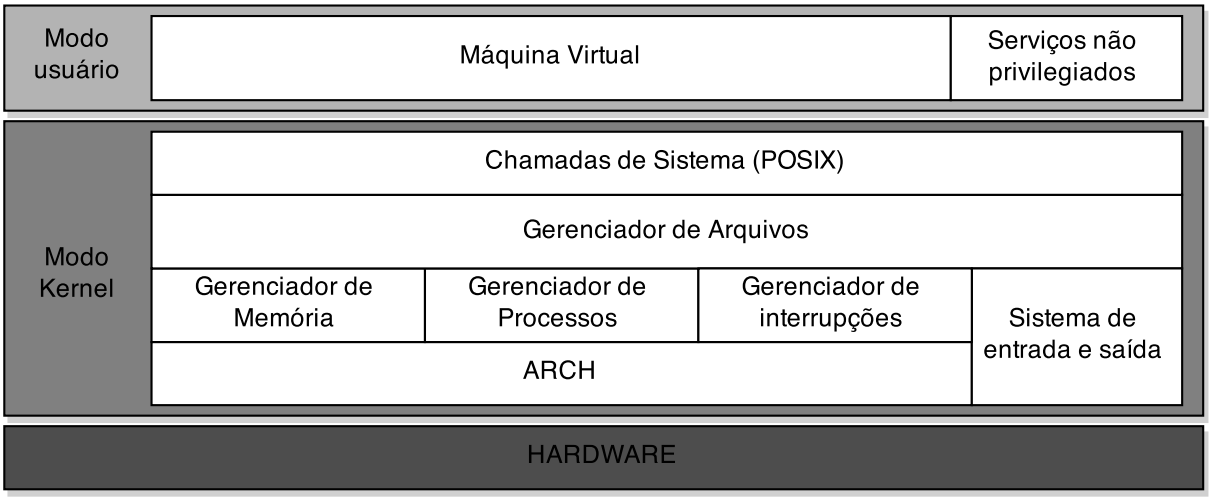


Figura 3.14: Organização em modo usuário.

Na Figura 3.14 podemos visualizar o relacionamento entre os módulos da arquitetura em modo usuário. Como podemos notar, a máquina virtual é executada em modo não privilegiado. Toda a comunicação entre o *kernel* e a máquina virtual é feita por chamadas de sistemas. Qualquer tentativa de acessar o *hardware* diretamente resulta em uma exceção.

Essa arquitetura traz como principal vantagem o desacoplamento do *kernel*, já que os programas compilados podem ser executados em qualquer sistema compatível com o POSIX. Além disso, torna possível o uso de diversas ferramentas de depuração e testes disponíveis, simplificando o desenvolvimento da máquina virtual.

A principal desvantagem desta arquitetura é a ineficiência. Infelizmente, a máquina virtual não tem como acessar componentes de *hardware* diretamente, ficando restrita ao gerenciamento feito pelos gerentes e das bibliotecas portadas. Além disso, aplicações muito complexas podem requerer um *kernel* mais robusto e maduro, como Linux.

3.8.3 Sistema em modo híbrido

O sistema em modo híbrido herda as principais vantagens das duas arquiteturas anteriores. Sem dúvida alguma é a arquitetura que possui a melhor relação entre desempenho, portabilidade e segurança.

Nessa arquitetura a máquina virtual é dividida em duas camadas, como podemos visualizar na Figura 3.15. Os módulos menos privilegiados passam a ser executados em modo usuário e os mais privilegiados em modo *kernel*. Para que isso seja possível, o

hypervisor deve implementar a sua própria versão do *kernel* - da mesma forma que foi apresentado na Seção 3.8.1 - fornecendo um conjunto de chamadas de sistemas estendido para as camadas superiores[17]. Dessa forma, programas em modo usuário podem ter acesso a recursos específicos do *hypervisor*.

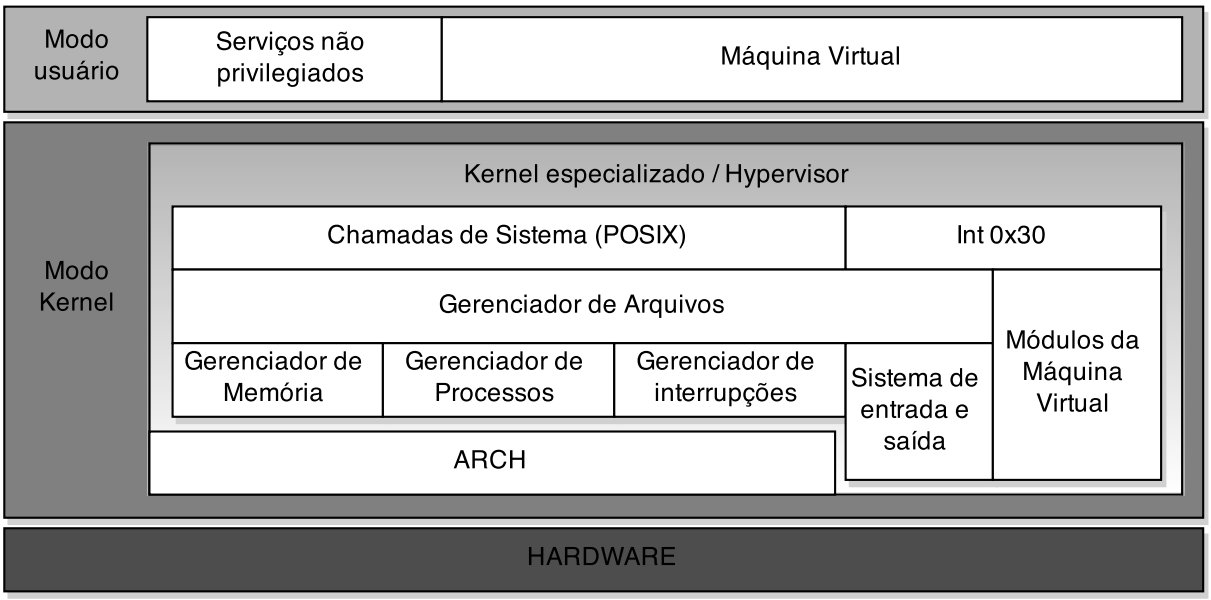


Figura 3.15: Organização em modo híbrido.

Os módulos menos privilegiados podem ser construídos fazendo uso das chamadas de sistemas compatíveis com o POSIX, herdando os principais benefícios da arquitetura em modo usuário. Já o núcleo da máquina virtual, pode ser completamente construído em modo *kernel*, tendo acesso direto a recursos e configurações do *hardware*. A interrupção 0x30 beneficia as camadas em modo usuário, ao fornecer uma interface única de comunicação com os módulo privilegiados da VM.

A principal desvantagem dessa arquitetura é a complexidade. Neste modelo é necessário dividir o sistema em diversos módulos independentes, reimplementando as rotinas não desejadas. Contudo, acreditamos que este modelo é adequado a implementações especializadas, já que os módulos podem se adequar inteiramente a necessidade da máquina virtual, possibilitando a implementação de um *hypervisor* simples e funcional.

[17]Reservamos a linha 0x30 para este propósito.

Capítulo 4

Suporte ao desenvolvimento de máquinas virtuais.

Para simplificar o desenvolvimento do *kernel* e dos programas compatíveis com a nossa infraestrutura, criamos uma série de *scripts* para auxiliar no processo de configuração, compilação, execução e teste de programas.

A maior parte do processo de configuração é efetuado pelo *script* config.sh[1]. O config.sh compila e instala, todas as ferramentas utilizadas na geração dos binários nativos. Esse processo é necessário, pois optamos por utilizar um compilador personalizado e que fizesse uso de somente dos recursos oferecidos pelo nosso *kernel*, ao invés de utilizar o GCC padrão.

Ao final da configuração, o config.sh gera um *cross compiler*, resultante da combinação do GCC com a NewLib. Esse *cross compiler* é configurado para gerar binários compatíveis com o padrão i686-elf. Até o momento, o nosso *script* de compilação gera epenas binários para a arquitetura x86. Contudo, em teoria, o GCC e a Newlib, podem ser configurados para gerar binários para dezenas de arquiteturas diferentes [52], necessitando apenas a implementação de algumas funções dependentes de sistema.

Uma das vantagens de utilizar um *cross compiler* personalizado é simplificação do processo de compilação. Uma vez instalada todas as ferramentas, programas em C++ poderão ser compilados com o i686-elf-g++, já que todas funcionalidades e recursos incompatíveis já estarão desligados por padrão.

Incluso nos objetos que são utilizados na criação dos binários está a nossa implementação das rotinas de chamadas de sistemas e de alguns outros símbolos relevantes. Simplificando e escondendo toda a complicação existente no processo de criação de binários compatíveis com os nossos módulos.

Na criação do *kernel* alguns binários devem ser adicionados no processo de ligação.

[1] Até o momento este *script* é compatível apenas com o *Linux*.

Além disso, precisamos fazer uso de um conjunto específico de *flags*, para informar o compilador que não é necessário incluir certos recursos utilizados em programas hospedados. Essas configurações poderiam ter sido inseridas no *cross compiler*, automatizado o processo de geração de programas nativos, mas isso faria com que os binários gerados pelo i686-elf-gcc não fossem compatíveis com o *Linux*. Assim, optamos por utilizar o i686-elf-gcc como um compilador de programas *hosted* e prover as *flags* necessárias no momento da geração dos binários nativos.

Para simplificar o processo de compilação do *kernel* criamos um *makefile*, que faz chamadas a alguns programas de apoio, utilizados na compilação e execução dos binários. O nosso script de compilação varre todos os diretórios recursivamente procurando pelos arquivos .cpp, .s e .c e faz a compilação e a ligação conforme as regras apresentadas no *makefile*. A implementação do *kernel* da máquina virtual deve ser inserida em um aquivo kernel.cpp. Que por padrão fica no diretório src.

O processo de compilação é extremamente simples e consiste de uma simples execução do comando *make*. No futuro pretendemos simplificar ainda mais o processo, fornecendo bibliotecas estáticas que serão ser ligadas aos programas, ao invés de forçar que os programadores utilizem a nossa infraestrutura de compilação. Contudo, até o momento, a alternativa mais simples de compilar, testar e executar programas é fazer uso da nossa organização de diretórios.

4.1 Organização dos diretórios

O diretório principal é organizado em 11 pastas, conforme a Tabela 4.1. As pastas mais importantes são a inc e a src, que contém a implementação de todos os módulos apresentados no Capítulo 3. Essas duas pastas são varridas recursivamente na etapa de compilação. Dessa forma, as subpastas podem ser modificadas a depender da necessidade de cada projeto, já que a compilação não depende da organização do subdiretórios.

Na tentativa de organizar o processo de compilação, evitando a presença de arquivos binários nos diretórios dos códigos fontes, nosso *script* de compilação direciona todos os objetos, arquivos temporários e binários para pasta bin. A imagem .iso resultante da compilação também é criada neste diretório.

Outra pasta extremamente importante é fs, que contém o mapa dos diretórios que serão copiados para imagem final do sistema. Todos os arquivos e pastas inseridos no diretório fs terão uma referência equivalente no diretório raiz do *kernel*. A imagem é gerada por um programa utilitário presente na pasta util, o imgbuild. O imgbuild foi implementado utilizado o NFSII, como sistema de arquivos padrão. Esse programa recebe como parâmetro um diretório e gera como saída uma imagem que pode ser integrada aos programas na etapa de ligação.

Pasta	Descrição
bin	Local onde todos os binários e arquivos temporários gerados na compilação são salvos.
cross	Pasta que contém o código fonte do compilador e das bibliotecas que serão usadas na geração do *cross compiler*.
doc	Pasta que contém a documentação do projeto e dados relevantes, como artigos, links e manuais.
emu	Pasta que contém o código fonte dos emuladores de exemplo.
fs	Modelo do sistema de arquivos que será clonado para dentro da imagem do sistema.
inc	Pasta que contém os arquivos *headers*.
lib	Pasta que contém as bibliotecas utilizadas na elaboração do *kernel*.
mod	Pasta que contém módulos não relacionados ao *kernel*.
src	Pasta que contém o código fonte do projeto.
test	Pasta que contém as implementações dos testes.
util	Pasta que contém a implementação de programas utilitários, utilizados na compilação do sistema.

Tabela 4.1: Organização dos diretórios.

4.2 Compilação e execução de programas

No momento da primeira compilação, o *script* procura pela presença de alguns utilitários importantes no sistema, como o qemu e grub-mkrescue. Se as dependências não existirem, o *script* tenta baixar e instalar estes programas.

Uma vez que todos os programas necessários estão instalados, o *script* de compilação cria uma imagem *iso* capaz de ser gravada em memórias *flash*, discos rígidos, cds e dvds. A imagem gerada também é compatível com diversas máquinas virtuais.

Para simplificar o trabalho de executar os programas nativos, inserimos uma regra *run* no *makefile*. Dessa forma, ao executar o comando "*make run*"o sistema configura o Qemu e inicializa uma VM com a imagem.iso já inserida no driver de cd virtual. Dessa forma, todo o processo de compilação e execução foi completamente automatizado.

Apesar da nossa infraestrutura ainda está em estágios iniciais, acreditamos que o suporte dado atualmente cumpre o papel de simplificar o desenvolvimento de programas nativos. Todo o processo de compilação foi abstraído, deixando o trabalho de configurar e ligar todos os módulos, uma tarefa da nossa infraestrutura de compilação.

4.3 Infraestrutura para testes

Sem dúvida alguma, o maior problema encontrado durante as fases inicias deste projeto foram os testes. É extremamente complexo testar módulos do *kernel*, uma vez que eles necessitam ser executado diretamente sobre o *hardware*.

Como a maioria dos módulos não são capazes de serem executados em modo usuário, em virtude do acesso a funções privilegiadas, as formas convencionais de depuração não podem ser utilizadas. Dessa forma, fizemos uso de diversos emuladores para checar o funcionamento e a compatibilidade entre os módulos do sistema. Dentre os emuladores utilizados, podemos citar: Qemu [21], Bochs [6], VirtualBox [29], VirtualPc [18], vmware Player [30], Parallels Desktop [24] e o KVM.

Apesar de reduzir o tempo de detecção de erros consideravelmente, fazer testes usando apenas emuladores não é suficiente. Em máquinas virtuais é comum a presença de simplificações no *hardware* emulado, na tentativa de elevar o desempenho. Isso pôde ser comprovado durante o desenvolvimento dos *drivers*, onde detectamos que alguns recursos de *hardware* não são emulados completamente, tornado algumas características raramente utilizadas inacessíveis. Em alguns casos, detectamos que erros específicos só eram visíveis nos *hardwares* reais. Assim, fizemos uso de ao menos uma dezena de computadores com configurações e fabricantes diferentes, para melhorar a confiabilidade dos testes.

Todo o processo de compilação e testes do sistema foram automatizados. Ao compilar o sistema, o script de compilação gera uma imagem do *kernel*, que é compatível com o padrão *multiboot* e que pode ser gravada em qualquer mídia inicializável.

Infelizmente, na nossa infraestrutura não existe um suporte para RTTI[2] ou STDlib[3]. Dessa forma, no estágio atual de desenvolvimento, bibliotecas que fizerem uso de recursos do modo usuário ou de recursos ainda não inicializados, não poderão ser executadas em modo *kernel*. Para que um programa rode nativamente, todo o suporte dependente de bibliotecas e serviços do modo usuário deve estar presente em modo *kernel*. Além disso, operações de exceções não podem ser utilizadas, uma vez que o *kernel* não provê um suporte para RTTI.

Um outro problema é que, para executar um teste, primeiro é necessário alocar diversos recursos de *hardware*. Em alguns casos, essa tarefa pode gerar um encadeamento complexo de dependências. Por exemplo, para alocar memória para um módulo, pode ser necessário carregar diversos outros módulos. A presença dessas dependências pode gerar comportamentos não desejáveis durante os testes, como a influência da implementação de outros módulos no resultado. Dessa forma, quanto menos recursos o *framework* de teste utilizar, melhor será a confiabilidade do resultado e mais simples será o trabalho de

[2] Acrônimo para *Run-Time Type Information*
[3] Acrônimo para *Standard Library*

detectar erros.

Infelizmente não fomos capazes de encontrar um *framework* de testes que atendesse aos nossos requisitos. Assim, paralelo ao desenvolvimento desse projeto, construímos um *framework* para testes de unidade. Escolhemos esta alternativa principalmente em virtude da incompatibilidade de certas camadas do *kernel* com as principais ferramentas de teste do mercado.

```
TEST( Memory_AllocateAndDeallocate )
  uintptr_t pages[BUFFER_SIZE] = {0};

  Memory m;
  size_t freeSize = m.getFreeSize();

  REQUIRE( freeSize >= MEMORY_MINIMUM_REQUIREMENT )

  SUBTEST( RandomAllocationAndDeallocation )
    srand( rand() );

    for ( unsigned int i = 0; i < 80000; i++ ) {
      int pos = rand() % BUFFER_SIZE;

      if ( pages[pos] == 0 ) {
        pages[pos] = m.allocate( ( pos % 5 ) + 1 );
        REQUIRE( pages[pos] != 0 );
        REQUIRE( !(((uintptr_t) pages[pos]) & ( MEMORY_PAGE_SIZE - 1 )) );
      } else {
        REQUIRE( m.deallocate( pages[pos], ( pos % 5 ) + 1 ) );
        pages[pos] = 0;
      }
    }
  END

  for ( unsigned int i = 0; i < 4; i++ )
    CALL( RandomAllocationAndDeallocation )

  for ( unsigned int i = 0; i < BUFFER_SIZE; i++ ) {
    if ( pages[i] )
      REQUIRE( m.deallocate( pages[i], ( i % 5 ) + 1 ) );
  }

  REQUIRE( m.getFreeSize() == freeSize );
END
```

Figura 4.1: Testando o alocador de memória.

Algumas camadas, como a que implementa o sistema de arquivos, poderiam ser testadas com qualquer *framework* disponível. Contudo, os *drivers*, ou módulo que necessitem de acesso privilegiado, poderiam requerer uma implementação alternativa extremamente complexa, para que fosse possível testar a implementação de forma satisfatória. Além

disso, a duplicação da implementação, para se adequar aos testes, poderia gerar resultado não tão precisos, já que teríamos que simular o comportamento de certas camadas inferiores durante os testes dos módulos privilegiados.

A infraestrutura de testes projetada é bastantes simple. Tomamos como exemplo o funcionamento do CATCH, um *framework* para testes de unidades que é capaz de operar com um número bem reduzido de dependências [7]. A nossa implementação é composta por apenas 4 macros: CALL, REQUIRE, TEST e SUBTEST.

No trecho de código apresentada na Figura 4.1 podemos visualizar o teste de um módulo privilegiado. No teste verificamos se o alocador e o desalocador estão operando sobre os blocos de páginas corretamente. Usamos uma técnica de teste aleatório para reforçar a confiabilidade deste módulo[4].

Acreditamos que a nossa implementação dos testes, apesar de minimalista, é uma grande contribuição para o *kernel* e todos os serviços que executem em modo privilegiado. Com o uso do nosso *framework*, todos os testes passam a ser automatizados. Uma vez compilado o projeto, um *script* gera uma imagem test.iso, que pode ser utilizada em qualquer computador pessoal. Os resultados dos testes podem ser configurados para serem enviados pela UART ou vídeo. Dessa forma, podemos testar todos os *drivers* e módulos do sistema sem a preocupação com concorrência de recursos ou dependência externa. Além disso, com o uso da nossa ferramenta de testes podemos detectar de forma direta quais módulos apresentam incompatibilidade com o *hardware*.

No futuro pretendemos aprimorar ainda mais o suporte para depuração e testes. Está em nossos planos inserir um suporte para o GDB, que poderá ser utilizado via uma conexão UART. Acreditamos que com o GDB será possível uma análise dos estado dos registradores no momento da execução, elevando a confiabilidade simplificando o trabalho de detecção e correção de erros.

[4]Note que estamos testando indiretamente se o sistema foi capaz de detectar a memória disponível, algo que não seria possível de ser feito em modo usuário.

Capítulo 5

Estudos de Caso

Como descrito no Capítulo 3, a nossa infraestrutura é capaz de trabalhar com diferentes níveis de privilégio. Assim, neste capítulo apresentamos alguns exemplos de como a nossa infraestrutura pode ser utilizada. Nas próximas seções discutiremos com detalhes alguns estudos de casos que demonstram o funcionamento desta arquitetura[1].

5.1 Máquina virtual hospedada

Como descrito no Capítulo 3 o comportamento de uma máquina virtual hospedada é similar ao de um aplicativo em modo usuário. Nessa organização a máquina virtual possui diversas restrições, não sendo capaz de acessar recursos privilegiados diretamente.

Buscando demonstrar a implementação de uma máquina virtual hospedada, portamos um emulador de 8086 capaz de executar diversos sistemas operacionais antigos, incluindo o DOS, Unix e o Windows 3.11. Este emulador, conhecido por tiny8086 [41], foi desenvolvido completamente em C por Adrian Cable.

Escolhemos o tiny8086 pelo fato de ser um emulador extremamente simples e compacto, algo que facilitou a portabilidade para nossa infraestrutura. Pequenas alterações foram realizadas no código do tiny8086 já que a versão atual da nossa infraestrutura não possui suporte para SDL[2]. Contudo, graças à compatibilidade com o padrão POSIX, o esforço necessário para portar e compilar esse aplicativo foi amortizado, requerendo apenas a substituição de algumas chamadas de sistemas por alternativas compatíveis com o nosso conjunto de *syscalls*[3].

Apesar da simplicidade desta aplicação de exemplo, o binário gerado faz uso de 21

[1] Todos os exemplos apresentados neste capítulo podem ser encontrados na pasta "emu", presente no diretório principal.

[2] Simple Direct Media Layer.

[3] O código do emulador, incluindo todas as alterações podem ser encontrados no arquivo tiny8086.cpp.

chamadas de sistemas diferentes. Algumas dessas rotinas exigem a implementação de dezenas de serviços associados, como por exemplo a *syscall open*, que necessita de um suporte para mais de 18 *flags* diferentes[15].

Garantir compatibilidade intrínseca entre sistemas operacionais é uma tarefa extremamente complexa. Como mencionado anteriormente, este trabalho não busca prover uma implementação alternativa do Linux, e sim apresentar uma infraestrutura mínima que facilite a interoperabilidade de ferramentas, bibliotecas e programas compatíveis com o POSIX. Portanto, somente as funcionalidades fundamentais para a execução de programas são suportadas pelo módulo de chamadas de sistemas.

O tiny8086 pode ser compilado facilmente utilizando o nosso *cross-compiler*, com o comando *i686-elf-gcc*. Para garantir a interoperabilidade entre sistemas compatíveis com o POSIX, o nosso *cross-compiler* liga estaticamente o binário a um objeto responsável por efetuar a inicialização do programa. Esse objeto, que deve ser obrigatoriamente ligado ao programa, é uma implementação alternativa do crt0, que contém as rotinas responsáveis por efetuar as configurações necessárias, antes da execução do programa.

```
#include <nkii.h>

int main() {
  System system;
  SysCalls syscalls;
  Memory memory;
  Scheduler scheduler;
  Storage storage;
  NFSII fs( storage );
  Video video;
  UART uart( COM0 );

  system.install( video );
  system.install( syscalls );
  system.install( scheduler );
  system.install( memory );
  system.install( fs );
  system.install( storage );
  system.install( uart );

  system.setDefaultInput( uart );
  system.setDefaultOutput( uart );
  system.setDefaultError( video );

  system.exec( "tiny8086" );
  system.start();
  return 0;
}
```

Figura 5.1: Exemplo de um *kernel* responsável por prover os serviços necessários para execução de binários.

Como o nosso compilador, por padrão, liga automaticamente o programa com uma implementação alternativa do crt0, que é compatível com o *Linux*, o binário gerado pode ser executado diretamente em qualquer distribuição que utilize o *Linux* como *kernel*. Portanto, para executar esse programa em nossa infraestrutura, precisamos de um *kernel* simples, que ofereça os recursos mínimos necessários para carregar o nosso binário para memória.

Na Figura 5.1 apresentamos um *kernel* de exemplo, projetado para carregar o tiny8086 para memória. O *Kernel* apresentado faz uso da nossa infraestrutura e é extremamente compacto, provendo somente os recursos necessários para execução de aplicativos de forma mono tarefa.

Como podemos notar, o comando system.exec("tiny8086") é responsável por executar o tiny8086. Por padrão, o *script* de compilação montará uma imagem com todos os arquivos e pastas presentes no diretório fs. Desta forma, para que o *kernel* produzido seja capaz de encontrar este aplicativo, é preciso copiar o binário gerado pelo i686-elf-gcc, incluindo o arquivo *input*, para pasta fs, antes de compilarmos o *kernel* com o comando *make*.

Na Figura 5.2 podemos visualizar algumas imagens retiradas durante a execução deste emulador em uma máquina virtual. Nesse exemplo toda interação com o sistema é feita via *UART*, simulando um gerenciamento remoto. O Qemu foi utilizado na simulação para mapear o dispositivo *UART* para o terminal.

```
 type HELP to get support on commands and navigation

A:\>dir
 Volume in drive A is FREEDOS1
 Volume Serial Number is 4559-120D
 Directory of A:\

KERNEL   SYS        45,450  04-07-2012  8:13a
COMMAND  COM        66,090  12-10-2003  7:49a
CONFIG   SYS           734  02-13-2014  9:06p
AUTOEXEC BAT           894  01-19-2014  2:04a
FREEDOS             <DIR>  10-07-2006 10:56a
APPINFO  TGZ        11,021  10-11-2006  3:00p
QUITEMU  COM             5  06-19-2013  4:35p
ALLEYCAT EXE        55,067  08-26-1998  2:52p
        7 file(s)        179,261 bytes
        1 dir(s)          57,344 bytes free

A:\>
```

Figura 5.2: Exemplo de um emulador hospedado.

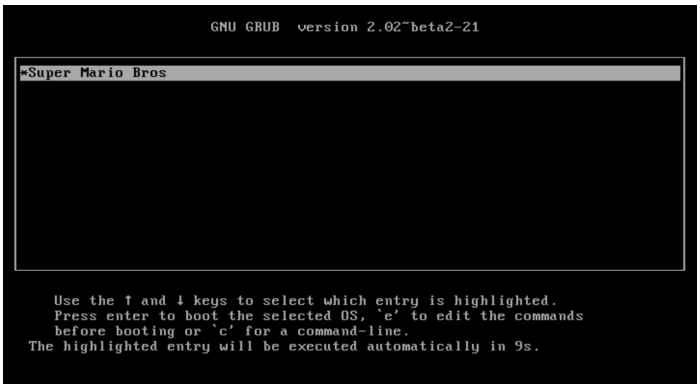
5.2 Uma máquina virtual de sistema nativa

Diferente da organização hospedada, onde a aplicação deve ser compilada separadamente e em seguida devemos prover um *kernel* com os recursos necessários para a execução do

aplicativo. Em uma arquitetura nativa a aplicação é o próprio *kernel*.

Para demonstrar o funcionamento desta organização, implementamos um emulador do processador Ricoh 2A03 diretamente no *kernel*[4]. O emulado faz uso dos *drivers* de vídeo e teclado para permitir a interação com o usuário.

O código do interpretador, sistema de interrupções, do BIOS e dos periféricos de entrada, foram retirados do aplicativo nesemu. O nesemu é um emulador extremamente compacto e preciso, desenvolvido por Joel Yliluoma. Com o uso da implementação de Joel conseguimos rodar diversos jogos do Nintendo de forma nativa, sem o auxílio de nenhum sistema operacional. Algumas características do código original tiveram que ser modificadas, para que o emulador pudesse se adequar melhor às necessidades de um aplicativo nativo, mas em essência não fizemos nenhuma adição significativa ao código original.



(a) Tela do grub.

(b) Tela inicial do emulador.

Figura 5.3: Exemplo de um emulador nativo.

[4]O código do emulador pode ser visualizado no arquivo nativeemu.cpp

O nesemu apresenta um suporte rústico para uma função de *autoplay*, que permite fazer uso de arquivos .fmv para jogar automaticamente. Usamos esta funcionalidade na versão nativa, permitindo um exercício de diversos módulos do sistema de forma automática. Na Figura 5.3 podemos visualizar uma imagem extraída durante a execução do sistema.

Esse emulador se mostrou extremamente portável e conseguimos executá-lo em diversos computadores diferentes, através de uma imagem gravada em um *pendrive*. Além disso, fomos capazes de instalar esta máquina nativa diretamente em disco, juntamente com o Linux.

Um fato interessante a respeito desse exemplo é que, até o exato momento, essa aplicação pode ser considerada como a primeira implementação de um emulador de console de vídeo game nativo, executando sobre um processador x86. No futuro com um suporte para SDL e *drivers* mais avançados seremos capazes de executar emuladores bem mais complexos e de consoles mais modernos.

5.3 Estatísticas gerais

Na Tabela 5.1 descrevemos as principais características deste projeto. Tomamos como base a organização descrita na Seção 5.2, onde apresentamos uma máquina virtual nativa para emular o *hardware* de um console de video game.

Informações gerais.	
Número total de linhas	7570 linhas
Número de linhas de código independentes de arquitetura (C/C++)	5941 linhas
Número de linhas de código dependentes de arquitetura	1458 linhas
Número total de linhas de código de configuração	171 linhas
Número total de linhas de código de teste	857 linhas
Total de arquivos da infraestrutura	104 arquivos
Total de arquivos de todo o projeto	106254 arquivos
Tamanho total em disco	2.4 GB

Tabela 5.1: Estatísticas gerais do projeto.

No cálculo das linhas, não levamos em consideração os comentários. Contamos somente as linhas que contém código. Como podemos verificar, aproximadamente 21.5% da nossa implementação é dependente de arquitetura. Na prática, a maior parte desse código está diretamente relacionado à configurações de *hardware*, principalmente com o suporte para

drivers. Logo, dependendo da arquitetura alvo e dos requisitos da máquina virtual, esse valor pode ser reduzido consideravelmente.

Como fizemos uso de diversas ferramentas livres para auxiliar no desenvolvimento da nossa pesquisa, o projeto completo ocupa mais de 2.4 GB[5]. Contudo, a nossa infraestrutura é composta por apenas 104 arquivos, que representa 0.097% de todo o projeto. A maior parte do código serve para compilar o nosso *cross compiler* integrado com a newlib.

Em relação ao tempo de inicialização, a nossa infraestrutura conseguiu executar a máquina virtual descrita na Seção 5.2 em aproximadamente 3 segundos - contando o tempo do *boot* até a tela inicial do aplicativo. Na Tabela 5.2 descrevemos algumas etapas intermediárias com mais precisão.

Tempo de inicialização	
Carregamento	Menos de 1 segundo
Configuração	Menos de 1 segundo
Tela inicial do emulador	Menos de 2 segundos
Tempo total	Aproximadamente 3 segundo

Tabela 5.2: Tempo total de inicialização do emulador nativo.

Como na inicialização do sistema não temos como medir o tempo com exatidão, já que a configuração do *timer* é feita durante a execução da camada ARCH, optamos por cronometrar o tempo de forma manual. Apesar de a nossa medição não ser tão precisa, acreditamos que seja suficiente para ter uma ideia da percepção sentida pelo usuário ao executar o aplicativo pela primeira vez.

A maior parte do tempo é gasto na detecção dos dispositivos e no carregamento da imagem do sistema para memória, já que grande parte do processo é feito através de uma espera ocupada, pois inicialmente o sistema é executado de forma monotarefa.

Nas Tabelas 5.3 e 5.4 descrevemos o tamanho da imagem do sistema em disco. Contudo, na Tabela 5.3 enfatizamos o tamanho antes da ligação estática, que é necessária para gerar o objeto final. Optamos por separar as duas tabelas por um único motivo: como ligamos a infraestrutura estaticamente com a libc, libstdc++ e a libm, a maior parte do binário final é composto por bibliotecas que foram ligados à nossa infraestrutura.

[5]Levando em consideração os arquivos binários.

Tamanho dos objetos antes da ligação estática	
Máquina virtual	0.5 MB
Disco virtual	4.7 MB
libc.a	3.4 MB
libstdc++.a	8.6 MB
libm.a	1.3 MB

Tabela 5.3: Tamanho dos objetos antes da ligação.

Como podemos notar na Tabela 5.3 os objetos que compõem a máquina virtual ocupam aproximadamente 0.5 MB. Contudo, após a ligação estática o tamanho total da imagem sobe para 8.3 MB, como podemos visualizar na Tabela 5.4. Além disso, a imagem virtual do disco, que contém o sistema de arquivos montado, é ligado juntamente com o binário final, agregando mais 4.7 MB ao tamanho total. Dessa forma, temos que o binário final - formado pela união da libc.a, libm.a, libstdc++, o disco virtual e as áreas de memória reservadas estaticamente - ocupa aproximadamente 13 MB. Os 8 MB restantes são reservados para o GRUB.

Tamanho da imagem final	
Grub	8.0 MB
Máquina virtual (módulos + aplicação + libc.a + libm.a + libstdc++)	8.3 MB
Disco virtual	4.7 MB
Alocação estática da pilha	64.0 KB
Alocação estática da *heap*	64.0 KB
Tamanho da image de CD/DVD (disk.iso)	˜21.0 MB

Tabela 5.4: Tamanho final da imagem.

Em relação a memória, o binário completo apresentado na Seção 5.2 ocupa 12 MB de RAM[6]. Conforme apresentado na Tabela 5.5, a máquina virtual, incluindo as bibliotecas, ocupa aproximadamente 7.1 MB. Contudo, na prática, a nossa infraestrutura consome pouco mais de 81 KB [7]. A maior parte dos 7.1 MB é gasto para armazenar o conteúdo das bibliotecas ligadas estaticamente.

[6]Sem contar as áreas de memória reservadas.
[7]Sem contar a implementação da máquina virtual.

Tamanho da máquina virtual na RAM.	
Espaço reservado para modo o 8086	1 MB
Tamanho da pilha	64.0 KB
Tamanho da *heap*	64.0 KB
Espaço desperdiçado com alinhamento	$<$ 4.0 KB
Sistema de arquivos em memória	4.7 MB
Máquina virtual (módulos + aplicação + libc.a + libm.a + libstdc++)	˜7.1 MB
Tamanho total	13.0 MB

Tabela 5.5: Consumo de memória.

Mesmo gastando 13 MB de RAM, o consumo de memória pode ser considerado baixo quando comparado com a quantidade de memória utilizada por um aplicativo em um sistema de propósito geral. O SliTaz [26], por exemplo, que é uma das distribuições mais leves atualmente, requer pelo menos 32 MB de RAM para executar em ambiente gráfico. Conseguimos com 13 MB RAM prover um ambiente *POSIX* mínimo para executar uma máquina virtual com suporte gráfico.

Capítulo 6

Conclusão

Esse projeto foi concebido para dar suporte à execução de máquinas virtuais nativas. Com o avanço da nossa pesquisa, detectamos que esse sistema poderia ser aplicado em diversas áreas correlatas. Acreditamos que a infraestrutura desenvolvida poderá ser utilizada desde a criação de programas nativos até o desenvolvimento de novos sistemas operacionais.

Graças à utilização de uma linguagem orientada a objetos, será possível especializar os diversos módulos do *kernel* para que eles se adequem ao funcionamento da máquina virtual. Essa característica permitirá uma maior integração entre toda a arquitetura, possibilitando a construção de *kernels* extremamente especializados. Além disso, essa característica permitirá uma análise da influência dos algoritmos de gerenciamento de recursos no funcionamento da máquina virtual, abrindo espaço para otimizações mais avançadas, como o uso de técnicas de *pass through*.

A adoção do padrão POSIX permitiu que programas em modo usuário pudessem ser recompilados e executados de forma nativa. Em nossos testes, conseguimos executar diversos aplicativos simples, sem grandes problemas. Essa funcionalidade também simplificará a depuração e testes de módulos, em virtude da compatibilidade binária existente entre o nosso *kernel* e o Linux.

Além disso, a nossa organização em módulos independentes possibilitou a criação de diversas variações da mesma arquitetura. Dependendo da aplicação, uma máquina virtual pode ser construída em diferentes níveis de privilégio. Essa característica permite que esse projeto seja utilizado, não só para o desenvolvimento de máquinas virtuais nativas, mas também, para o desenvolvimento de máquinas virtuais hospedadas e e híbridas.

Paralelamente à elaboração dessa infraestrutura, construímos um ecossistema para desenvolvimento de máquinas virtuais, incluindo um *framework* para testes e ferramentas que automatizam diversas tarefas, incluindo a compilação.

Desta forma, agradecemos à FAPESP e o CNPq, pelo apoio financeiro, e à UNICAMP, por fornecer a infraestrutura e a base teórica necessária para a elaboração deste trabalho.

Referências Bibliográficas

[1] Ahci. http://lxr.free-electrons.com/source/drivers/ata/ahci.c.

[2] Android: a open source project. http://source.android.com.

[3] Android-x86: run android on your pc. http://www.android-x86.org/.

[4] Apple rosetta.

[5] Ata. http://wiki.osdev.org/IDE.

[6] Bochs: the open source ia-32 emulation project. http://bochs.sourceforge.net/.

[7] Catch. https://github.com/philsquared/Catch.

[8] Citrix. http://www.citrix.com/products/xenserver/overview.html.

[9] cygwin: get that linux feeling on windows. https://www.cygwin.com/.

[10] Exokernel. http://pdos.csail.mit.edu/exo/.

[11] How to use huge pages to improve application performance on intel® xeon phi™ coprocesso. https://software.intel.com/sites/default/files/Large$_p ages_m ic_0 .pdf$.

[12] Jnode os. http://www.jnode.org/node/68.

[13] Jx os. http://www4.cs.fau.de/Projects/JX/publications/jx-usenix.pdf.

[14] The kaffe virtual machine. http://www.kaffe.org/.

[15] Linux manual: syscall open. http://man7.org/linux/man-pages/man2/open.2.html.

[16] Linux programmer's manual: elf. http://man7.org/linux/man-pages/man5/elf.5.html.

[17] Linux programmer's manual: syscalls. http://www.linux.com/learn/docs/man/syscalls2.

[18] Microsoft virtual pc. http://www.microsoft.com/en-us/download/details.aspx?id=3702.

[19] Newlib. http://sourceware.org/newlib/.

[20] Newlib documentation. https://sourceware.org/newlib/libc.htmlSyscalls.

[21] Open source processor emulator. http://wiki.qemu.org/Main$_Page$.

[22] Oraclevm. http://www.oracle.com/us/technologies/virtualization/oraclevm/specifications/.

[23] Osv: the operating system designed for the cloud. https://www.osv.io/.

[24] Parallels desktop. http://www.parallels.com/br/products/desktop/?src=rgclid=CISftIeUocQCFU8Q

[25] Seagate: Ata interface reference manual. hftp://ftp.seagate.com/acrobat/reference/111-1c.pdf.

[26] Slitaz gnu linux. http://www.slitaz.org/.

[27] System mmu whitepaper. http://www.arm.com/files/pdf/system-mmu-whitepaper-v8.0.pdf.

[28] V8 - google engine. https://developers.google.com/v8/.

[29] Virtual box. https://www.virtualbox.org/.

[30] Vmware player. http://www.vmware.com/products/player.

[31] Wine: Wine is not an emulator. https://www.winehq.org/.

[32] Xenproject. http://www-archive.xenproject.org/files/Marketing/HowDoesXenWork.pdf.

[33] Standard for information technology - portable operating system interface (POSIX). Shell and utilities. Technical report, 2004.

[34] E. Azimzadeh, M. Sameki, and M. Goudarzi. Performance analysis of android underlying virtual machine in mobile phones. In *Consumer Electronics - Berlin (ICCE-Berlin), 2012 IEEE International Conference on*, pages 292–295, Sept 2012.

[35] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00)*, pages 1–12, New York, NY, USA, 2000. ACM.

[36] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, page 191, Washington, DC, USA, 2003. IEEE Computer Society.

[37] Carter Bays. A comparison of next-fit, first-fit, and best-fit. *Commun. ACM*, 20(3):191–192, March 1977.

[38] Nikhil Bhatia. Performance evaluation of amd rvi hardware assist. http://www.vmware.com/pdf/RVI$_p erformance.pdf$.

[39] Don Box and Ted Pattison. *Essential .NET: The Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[40] Thomas Wolfgang Burger. Intel® virtualization technology for directed i/o (vt-d): Enhancing intel platforms for efficient virtualization of i/o devices. https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices.

[41] Adrian Cable. 8086tiny. http://www.megalith.co.uk/8086tiny/.

[42] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. Automated reduction of the memory footprint of the linux kernel. *ACM Trans. Embed. Comput. Syst.*, 6(4), September 2007.

[43] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.

[44] Alfons Crespo, Ismael Ripoll, and Miguel Masmano. Partitioned embedded architecture based on hypervisor: The xtratum approach. In *EDCC*, pages 67–72, 2010.

[45] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization (CGO '03)*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Inc Dvanced Micro Devices. Amd-v nested paging. http://www.cse.iitd.ernet.in/ sbansal/csl862-virt/2010/readings/NPT-WP-1

[47] Dawson R Engler, M Frans Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 251–266. ACM, 1995.

[48] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03)*, pages 278–288, New York, NY, USA, 2003. ACM.

[49] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for OS and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[50] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In Karin Petersen and Willy Zwaenepoel, editors, *OSDI*, pages 137–151. ACM, 1996. Operating Systems Review 30, Special Issue, October 1996.

[51] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. The Flux OS toolkit: Reusable components for OS implementation. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 14–19, Cape Cod, MA, May 1997. IEEE Computer Society.

[52] Free Software Foundation. Status of supported architectures from maintainers' point of view. https://gcc.gnu.org/backends.html.

[53] Tom R. Halfhill. Transmeta breaks x86 low-power barrier. *Microprocessor Report*, 14(2), 2000.

[54] Raymond J. Hookway and Mark A. Herdeg. Digital fx!32: combining emulation and binary translation. *Digital Tech. J.*, 9(1):3–12, 1997.

[55] Randall Hyde. *Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level*. No Starch Press, San Francisco, CA, USA, 2006.

[56] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '03, pages 6–6, Berkeley, CA, USA, 2003. USENIX Association.

[57] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[58] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX*

Annual Technical Conference (USENIX ATC 14), pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[59] Paul Klint. Interpretation techniques. *Software — Practice & Experience*, 11(9):963–973, September 1981.

[60] Kevin Krewell. Transmeta gets more efficeon. *Microprocessor Report*, 17(10), 2003.

[61] David Cowperthwaite Kun Tian, Yaozu Dong. A full gpu virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, June 2014. USENIX Association.

[62] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[63] Linhares. Neutrino os: Uma abordagem original para a construção de sistemas operacionais. https://projetonos.wordpress.com/.

[64] Jiuxing Liu, Wei Huang, Bülent Abali, and Dhabaleswar K Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 29–42, 2006.

[65] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation (PLDI '05)*, pages 190–200, New York, NY, USA, 2005. ACM.

[66] F.B. Machado and L.P. Maia. *Arquitetura de sistemas operacionais*. Livros Tecnicos e Cientificos, 1996.

[67] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. An overview of the xtratum nanokernel. In *Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2005.

[68] Andreas Jaeger Mark Mitchell Michael Matz, Jan Hubicka. Amd64 abi draft 0.99.6. http://www.x86-64.org/documentation/abi.pdf.

[69] Sun Microsystem. Java os. http://www.oracle.com/technetwork/java/index-135492.html.

[70] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT '08)*, June 2008.

[71] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices*, 42(6):89–100, 2007.

[72] Matt Pietrek. Peering inside the pe: A tour of the win32 portable executable file format. In *Msdn magazine: the Microsoft journal for developers*. MSDN Magazine, 1994.

[73] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications ACM*, 17(7):412–421, 1974.

[74] Marco Righini. Enabling intel virtualization technology features and benefits. http://www.intel.com.br/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf.

[75] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 6th edition, 2001.

[76] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java™ on the bare metal. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–151. ACM, 2005.

[77] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88. ACM, 2006.

[78] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications ACM*, 36(2):69–81, 1993.

[79] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[80] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[81] Aaron Tenenbaum. Memory utilization efficiency under a class of first-fit algorithms. In *Proceedings of the ACM 1980 Annual Conference*, ACM '80, pages 186–190, New York, NY, USA, 1980. ACM.

[82] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, June 2014. USENIX Association.

[83] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[84] Satyam B. Vaghani. Virtual machine file system. *SIGOPS Oper. Syst. Rev.*, 44(4):57–70, December 2010.

[85] P. Weisberg and Y. Wiseman. Using 4kb page size for virtual memory is obsolete. In *Information Reuse Integration, 2009. IRI '09. IEEE International Conference on*, pages 262–265, Aug 2009.

[86] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.

[87] Vmware® server 2: product datasheet. http://www.vmware.com/files/pdf/VMware-Server-2-DS-EN.pdf.