2
Conceitos Básicos

Neste capítulo, introduziremos alguns conceitos básicos que serão abordados no restante dessa dissertação.

2.1
Sistema Operacional

O conceito de sistema operacional (SO) é geralmente definido como a camada de *software* de um sistema computacional responsável por desempenhar dois papéis distintos:

Máquina virtual: fornece uma interface de abstração entre o usuário e o computador, possibilitando a utilização do *hardware* e de seus recursos sem a necessidade de conhecer as suas idiossincrasias.

Gerenciador de recursos: multiplexa o computador e seus recursos com segurança e eficiência, provendo, assim, o compartilhamento deste sistema (10, 11, 12).

Esta definição, contudo, é bastante imprecisa e vaga. Por exemplo, podemos utilizar essa mesma definição para vários outros tipos de *software*, como compiladores e interpretadores de linguagens de programação ou interpretadores de linha de comando, pois estes também oferecem uma interface de abstração e gerenciam recursos do computador. Essa definição também apresenta uma grande dispersão quanto ao seu escopo. Alguns autores costumam definir sistema operacional como a camada de *software* responsável por desempenhar os papéis de máquina virtual e gerenciador de recursos e que é executada no mais alto nível de privilégio do *hardware*, ou seja, o conceito usualmente chamado de *kernel* de sistema operacional. Outros autores costumam definir sistema operacional como o *software* responsável por esses dois papéis e mais todos os outros programas distribuídos junto com ele, como servidores gráficos, interpretadores de linha de comando, compiladores e bibliotecas; esse conceito é usualmente chamado de distribuição de sistema operacional. Existem também autores que costumam definir o sistema operacional como um *kernel* e mais alguns seletos aplicativos de sistema.

Não pretendemos propor aqui uma definição precisa para o conceito de sistema operacional. Contudo, apresentaremos uma definição mais estrita para esse conceito com o objetivo de evitar ambiguidades e equívocos. Desta forma, definimos o conceito de sistema operacional utilizado nesta dissertação como a camada de *software* responsável pelos papéis de máquina virtual e gerenciador de recursos e que *não pode ser evitada* pelos usuários do sistema computacional; estes usuários são definidos como usuários das aplicações, programadores das aplicações e as próprias aplicações em si. Entendemos por *"camada de software que não pode ser evitada"* todo o *software* que não pode ser implementado como parte integrante das aplicações e que também não podem ser dispensados por elas. Por exemplo, se para utilizar um dispositivo de disco rígido, uma aplicação não pode evitar o uso de sistemas de arquivos, então os sistemas de arquivos fazem parte da camada de *software* que não pode ser evitada. Portanto, nesse exemplo, os sistemas de arquivos são parte integrante do sistema operacional, sejam eles executados no *kernel* ou no nível de usuário.

Desta forma, o conceito de sistema operacional utilizado no restante desta dissertação abrange tanto código executado em modo privilegiado (*kernel*), quanto código executado em nível de usuário (servidores, bibliotecas e aplicativos de sistema em geral).

2.1.1 Problemas Conhecidos

Existe uma antiga percepção, que remete aos primeiros sistemas operacionais, de que as implementações existentes para esses sistemas apresentavam sérias inadequações, tanto sob o ponto de vista dos desenvolvedores quanto sob o dos usuários (1). Embora esta percepção de inadequação seja bastante antiga, ela não mudou muito ao longo dos anos; pelo contrário, ela tem estado constantemente presente ao longo da história dos sistemas operacionais até a atualidade (13, 3, 5). Essa inadequação é freqüentemente relacionada a insuficiência quanto aos seguintes fatores:

Flexibilidade: é a propriedade que determina a capacidade do sistema ser modificado ou contornado pelos usuários, uma vez que ele não pode ser evitado. Entendemos por *"contornar o SO"* emular o comportamento pretendido pelo usuário acima do SO. Por exemplo, uma aplicação poderia contornar a abstração dada pelo SO para o disco rígido, emulando a sua própria abstração para disco acima do SO. Sistemas operacionais tradicionais não possuem muitos recursos para serem modificados ou contornados e, como esses sistemas não podem ser evitados, eles precisam estar presentes em todas as aplicações. Conseqüentemente, as aplicações

não podem impedir que o sistema operacional impacte diretamente no funcionamento delas.

Confiabilidade: é a propriedade que determina a robustez do sistema, ou seja, o quanto esse sistema é suscetível a falhas e a comportamentos indesejáveis. Sistemas operacionais tradicionais são demasiadamente grandes e complexos por tentarem antever e implementar os requisitos de todas as aplicações. De uma forma geral, sistemas grandes e complexos são mais suscetíveis a falhas.

Desempenho: é a propriedade que determina a eficiência e rapidez da execução do sistema. Sistemas operacionais tradicionais, por serem demasiadamente grandes e estarem presentes em todas as aplicações, impõem uma alta e desnecessária sobrecarga. Além disso, como as aplicações tipicamente não possuem ou possuem pouca escolha, elas não são capazes de determinar completamente a maneira mais eficiente de atender aos seus requisitos.

Facilidade de desenvolvimento: é a propriedade que determina quão facilmente pode-se desenvolver, manter e evoluir a implementação do sistema. Sistemas operacionais tradicionais são implementados em linguagens de nível mais baixo, sem recursos modernos de desenvolvimento e depuração de código e com características bem diferentes da programação de aplicativos (e.g., assincronia excessiva, ausência de memória virtual). De uma forma geral, para desenvolver sistemas operacionais, desenvolvedores precisam de uma grande curva de aprendizado. Essa grande curva de aprendizado tem como conseqüência uma maior dificuldade em se desenvolver, manter e evoluir sistemas operacionais. Além disso, ela implica na pouca convidatividade para desenvolvedores de aplicações ou até mesmo usuários modificarem o sistema (quando isso é possível) para atenderem às suas próprias necessidades.

Nesta dissertação, nós corroboramos com a tese de que a insuficiência desses três últimos fatores — confiabilidade, desempenho e facilidade de desenvolvimento — decorrem ou estão diretamente relacionadas à falta de flexibilidade dos sistemas operacionais.

2.2 Sistemas Operacionais Extensíveis

Sistema operacional extensível é um abordagem de projeto de sistemas operacionais que surgiu no final da década de 1960 (1, 14), foi retomado durante a década de 1990 e tem sido extensivamente explorado em pesquisas

desde então (13, 15, 3, 4, 16, 17, 5). Nesta abordagem, é sustentado, particularmente, que é possível mitigar a inadequação causada pela insuficiência de flexibilidade, confiabilidade e desempenho dos sistemas operacionais utilizando extensibilidade. O argumento central defendido na abordagem de sistema operacional extensível é que sistemas operacionais de propósito geral *não podem* antever as necessidades de todos os seus usuários; portanto, é necessário que os seus usuários sejam capazes de adequar o comportamento do sistema operacional para atender a requisitos diferentes dos considerados durante o seu projeto (1, 6). Sistemas operacionais extensíveis são aqueles que permitem que seus usuários estendam o seu comportamento, referente ao papel de máquina virtual ou de gerenciador de recursos, para atenderem aos seus próprios requisitos.

Quando um sistema operacional permite extensões, ele automaticamente aumenta a capacidade dos usuários de adaptarem ou contornarem esse sistema para atenderem as suas próprias necessidades. Desta forma, extensibilidade proporciona o aumento da flexibilidade do sistema operacional. Ao possibilitarem que usuários adaptem ou evitem o sistema operacional, sistemas operacionais extensíveis possibilitam, em particular, que usuários ajustem o comportamento do sistema para obterem um maior desempenho em casos específicos. Por exemplo, o algoritmo de *cache* de disco que proporciona maior desempenho para um servidor *web* pode não ser o mesmo que apresenta maior desempenho para um sistema gerenciador de banco de dados. Desta forma, aumentar a flexibilidade do sistema operacional também pode conduzir a aumento de desempenho.

Sistemas operacionais que permitem a adaptação por usuários permitem também a redução do tamanho do sistema operacional em si, pois parte das funcionalidades do sistema pode ser implementada pelo usuário, através de extensões. Isto possibilita o foco do desenvolvimento do sistema operacional nas funcionalidades básicas e na proteção das extensões, em vez do foco no aumento de funcionalidades e na tentativa de atender as necessidades de todos os possíveis usos. Essa redução da implementação do sistema operacional possibilita o aumento da confiabilidade, pois, de uma forma geral, sistemas menores são mais fáceis de projetar, modularizar, testar, auditar e desenvolver e, também, são menos suscetíveis a erros e comportamentos inesperados.

2.2.1 Projeto e Implementação

Sistema operacional extensível é um conceito tão impreciso e vago quanto o próprio conceito de sistema operacional, pois praticamente todo sistema ope-

racional apresenta algum grau de extensibilidade, ou seja, permite adaptações de alguma forma (6). Contudo, estamos interessados na diferentes formas de provermos extensibilidade, em vez da definição precisa do conceito. Portanto, nesta seção, apresentaremos um conjunto de requisitos e características de projeto e implementação de sistemas operacionais extensíveis. Então, utilizaremos esses requisitos e características para analisar e classificar as diferentes formas de se prover extensibilidade em um sistema operacional.

Requisitos

Baseamos os requisitos de um sistema operacional extensível, listados nesta seção, no trabalho desenvolvido pelo grupo de pesquisa do sistema SPIN da Universidade de Washington (6). De acordo com esse trabalho, um sistema operacional extensível deve atender, de uma só vez, três requisitos básicos: incrementalidade, corretude e eficiência.

Incrementalidade

Incrementalidade é o requisito que trata da facilidade com que mudanças no comportamento do sistema operacional podem ser realizadas pelas extensões. Idealmente, pequenas mudanças no comportamento do sistema operacional devem ser possíveis de serem alcançadas através de pequenas quantidades de código.

As características que determinam a incrementalidade dos sistemas operacionais são a *granularidade* e a *programabilidade*, ambas abordadas na seção 2.2.1.

Corretude

Corretude é o requisito que trata da confiança e tolerância a falhas referentes às extensões. Esse requisito enuncia que uma extensão não deve comprometer completamente a integridade do sistema. Extensões podem violar a confiança do sistema operacional de várias maneiras. A seguir listamos algumas dessas possibilidades:

- entrar em um laço infinito;
- entrar em *deadlock*;
- falhar em bloquear ou desbloquear um recurso compartilhado;
- acessar uma interface usando parâmetros inapropriados;
- executar uma operação ilegal;
- tentar acessar um objeto que não existe mais.

As características que determinam como os sistemas operacionais podem garantir a corretude das extensões são *localização*, *autorização*, *proteção* e *arbitragem*, todas abordadas na seção 2.2.1.

Eficiência

Eficiência é o requisito que trata o desempenho das extensões. De acordo com esse requisito, o uso de uma extensão deve ter um *overhead* determinado pelo código da extensão, e não pela infra-estrutura que habilita a extensão. Várias características influem na eficiência, entretanto, as principais são *localização*, *proteção* e *programabilidade*. Abordamos todas elas na seção 2.2.1.

Características

Baseamos as características de um sistema operacional extensível, listadas nesta seção, principalmente no trabalho desenvolvido pelo grupo de pesquisa do sistema VINO da Universidade de Harvard (16). De acordo com esse trabalho, um sistema operacional extensível é caracterizado pelos seguintes pontos: *mutabilidade*, *localização*, *confiança/falha*, *tempo de vida*, *granularidade* e *arbitragem*. Entretanto, adicionamos os seguintes pontos baseados nas nossas próprias conclusões sobre estudos como (18), (6), (16) e (4): *autorização* e *programabilidade*. Além disso, utilizamos os termos *proteção* e *duração* para designar, respectivamente, os termos *confiança/falha* e *tempo de vida* para nos alinharmos de forma mais homogênea à terminologia utilizada em todos esses estudos. Mostramos o sumário dessas características na Tabela 2.2.1, assim, definindo uma taxonomia em sistemas operacionais extensíveis.

Características	Opções			
Mutabilidade	parametrizada	reconfigurável	extensível	
Localização	nível de usuário	nível intermediário	*kernel*	
Proteção	"boa vontade"	*hardware*	*software*	
Autorização	não-privilegiada	privilegiada		
Duração	aplicação	recurso	*kernel*	permanente
Granularidade	modular	procedural	procedural limitada	
Arbitragem	proibição	decisão separada	multiplexação	
Programabilidade	linguagem de sistema	linguagem de sistema limitada	linguagem de extensão	linguagem de domínio específico

Tabela 2.1: Taxonomia de Sistemas Operacionais Extensíveis

Mutabilidade

Mutabilidade é uma importante medida da flexibilidade de um sistema operacional. Ela determina *quando* (tempo de compilação ou execução) e *como* (parâmetros ou código) os usuários podem modificar o sistema. Sistemas operacionais podem ser classificados, não-exclusivamente, em *parametrizáveis*, *reconfiguráveis* ou *extensíveis* quanto aos seus graus de mutabilidade.

Sistemas operacionais *parametrizáveis* permitem a modificação do sistema através da configuração de parâmetros em tempo de compilação, o que possibilita a geração de sistemas modificados para atender a diferentes comportamentos. Esse tipo de mutabilidade se baseia na noção de que alguns sistemas podem ser adaptados para um tipo específico de aplicação. Nesse caso, o sistema operacional atua como um conjunto de ferramentas para a construção de sistemas especializados; como resultado temos um sistema customizado para os requisitos de determinada classe de aplicação. Contudo temos um sistema que não é dinamicamente adaptável. Um exemplo de sistema operacional parametrizável é Scout, desenvolvido pelo Departamento de Ciência da Computação da Universidade do Arizona, voltado para aplicações orientadas para comunicação (19). Além de Scout, muitos outros sistemas operacionais oferecem mutabilidade por configuração de parâmetros de compilação, principalmente sistemas operacionais livres, pela disponibilidade do código fonte (e.g., Linux, NetBSD, FreeBSD, OpenBSD).

Sistemas operacionais *reconfiguráveis* permitem a modificação do sistema através da configuração de parâmetros em tempo de execução, o que possibilita o ajuste dinâmico do comportamento do sistema. Esse tipo de mutabilidade se baseia na noção de que sistemas podem ser ajustados em tempo de execução para atender requisitos de aplicações específicas através de escolhas feitas dentre um conjunto pré-determinado (em tempo de compilação) de opções. Alguns exemplos de sistemas operacionais reconfiguráveis são Linux, por meio do *sysfs* (20), 4.4BSD e seus descendentes, por meio do *sysctl* (21) e Windows, por meio do *registry* (22).

Sistemas operacionais *extensíveis*, sob o ponto de vista da mutabilidade, permitem a modificação do sistema através da incorporação de novo código em tempo de execução, o que possibilita a adição de novos comportamentos ao sistema. Esse tipo de mutabilidade se baseia na noção de que é necessário permitir a adição de novo código no sistema operacional para atender a requisitos não previstos durante o projeto e implementação do sistema. Alguns exemplos de sistemas operacionais extensíveis são MS-DOS, Mach, Exokernel (13), SPIN (3), VINO (4), Linux, NetBSD.

Podemos avaliar as classificações da mutabilidade de sistemas operaci-

onais como uma escala incremental gradativa em relação a flexibilidade. Primeiro, temos o ajuste do sistema através da configuração de parâmetros em tempo de compilação (sistemas parametrizáveis), que proporciona alguma flexibilidade, mas permite apenas escolhas dentre um conjunto pré-determinado de opções. Em seguida, temos o ajuste de parâmetros em tempo de execução (sistemas reconfiguráveis), que proporciona mais flexibilidade do que o caso anterior por permitir a modificação do sistema durante a sua execução, mas ainda limita o ajuste à simples escolha dentre um conjunto de opções. Por fim, temos o ajuste do sistema através da incorporação de novo código em tempo de execução (sistemas extensíveis), que proporciona o maior grau de flexibilidade por permitir a modificação do sistema para atender a novos comportamentos.

Localização

Outra característica importante quanto à extensibilidade de um sistema operacional é a *localização* das extensões, isto é, *onde* as extensões serão acomodadas. Extensões podem ser acomodadas basicamente de três formas: no *kernel* (nível mais privilegiado do processador), no espaço de usuário (nível menos privilegiado do processador) ou em espaços intermediários (níveis de privilegio entre o modo *kernel* e o modo usuário, como, por exemplo, os *rings* 1 e 2 da arquitetura IA32).

Extensões são acomodadas no *kernel* tipicamente através de ligação de módulos ou carga de código. Este é o caso de sistemas que permitem carga de código, como Exokernel, SPIN e VINO; sistemas que permitem ligação de módulos, como Linux, NetBSD e Solaris; e sistemas que executam completamente (todos os processos) em modo *kernel*, como Singularity.

Extensões são acomodadas no espaço de usuário tipicamente através de bibliotecas ou processos servidores (e.g., *daemons*). Este é o caso de alguns sistemas baseados em *microkernel* mais tradicionais, como Mach, e sistemas como Exokernel, que acomodam abstrações do sistema operacional em espaço de usuário sob a forma de bibliotecas.

Extensões são acomodadas em espaços intermediários tipicamente através de servidores privilegiados. Este é o caso de alguns sistemas operacionais em camadas como THE (projetado por Dijkstra) (14) e CAL (projetado por Lampson) (1), de alguns sistemas baseados em *microkernel*, como MINIX, e alguns hipervisores como Xen (23). No caso de hipervisores, as extensões são sistemas operacionais completos.

Localização é uma característica que influi diretamente na flexibilidade, confiabilidade e desempenho do sistema operacional. Extensões acomodadas no espaço de usuário agregam menos flexibilidade e desempenho do que

extensões acomodadas no *kernel*, pois extensões no *kernel* podem ter acesso a estruturas internas. Por outro lado, extensões em espaço de usuário ou espaços intermediários agregam *hipoteticamente* mais confiabilidade ao sistema operacional, pois utilizam uma camada de isolamento mais rígida (as extensões executam em processos separados do *kernel*).

Proteção

Proteção é outra importante característica de extensibilidade que determina como protegemos o sistema das extensões para evitar danos, tanto provenientes de falhas quanto de códigos maliciosos, e garantir o funcionamento correto das extensões. Essa característica diz respeito aos graus de confiança e tolerância a falhas atribuídos às extensões, isto é, quanto as extensões podem modificar o sistema; quão confiáveis são as extensões para não corromperem o resto do sistema; e, em caso de falha, quanto o sistema será comprometido. A proteção pode ser realizada por três maneiras: "boa vontade", *hardware* ou *software*. A forma de proteção está diretamente relacionada com a localização das extensões, pois é a forma de proteção que na maioria das vezes determina onde as extensões serão alocadas.

Proteção por "boa vontade" é basicamente a ausência de proteção. Neste caso, as extensões são acomodadas diretamente no *kernel*, têm acesso irrestrito ao sistema (espaço de endereçamento de memória, dispositivos, etc) e, conseqüentemente, em caso de falha ou comportamento indesejado, podem comprometer completamente o sistema. Alguns exemplos de proteção por "boa vontade" são sistemas que utilizam módulos carregáveis de *kernel*, como Linux, FreeBSD, NetBSD e Solaris.

Proteção por *hardware*, como o próprio nome sugere, é a proteção baseada em recursos do computador em si, como proteção de memória utilizando a *MMU (Memory Management Unit)* e proteção quanto a falta de vivacidade (*liveness*) utilizando interrupções de *hardware*. Nesta forma de proteção, as extensões são acomodadas tipicamente fora do *kernel*, como processos localizados no espaço de usuário ou em espaços privilegiados intermediários. Desta forma, em caso de falha ou comportamento indesejado de uma extensão, apenas o processo que executa a extensão é comprometido. Contudo, outras partes do sistema que dependem dessa extensão podem ser afetadas indiretamente. Por exemplo, no caso de uma extensão que implementa um sistema de arquivos não funcionar corretamente, ela compromete todos os clientes desse sistema de arquivos. Alguns exemplos de proteção por *hardware* são sistemas *microkernel*, como MINIX e Mach.

Proteção por *software* é feita tipicamente utilizando-se recursos de lin-

guagem de programação, como compiladores, verificadores de código, interpretadores e máquinas virtuais. Por exemplo, proteção de memória pode ser garantida por *typesafety* e sincronização pode ser garantida por memória transacional ou monitores. Nessa forma de proteção, as extensões são tipicamente acomodadas dentro do *kernel*. Alguns exemplos de sistemas que utilizam essa forma de proteção são os sistemas que permitem que extensões executem dentro do *kernel*, como Exokernel, VINO, SPIN e Singularity.

Autorização

Autorização é uma característica que está intimamente ligada à proteção. Assim como a proteção, a autorização trata de meios de garantir a corretude das extensões. Autorização determina quem pode (ou seja, quem está autorizado para) instalar extensões. Classificamos a autorização de duas formas: não-privilegiada e privilegiada.

Autorização *não-privilegiada* (ou não-confiável) é a forma de autorização que permite que qualquer usuário possa instalar extensões no sistema. Nessa forma de autorização não é realizada nenhuma validação de quem está instalando a extensão. Assim, todos usuários do sistema podem instalar extensões. Um exemplo desse tipo de autorização é o MS-DOS, onde todos os usuários do sistema podem instalar extensões irrestritamente. Autorização não-privilegiada também é o caso de sistemas operacionais extensíveis da década de 90, como Exokernel, SPIN e VINO, que permitem a carga de código não-privilegiado no *kernel*. Garantir a proteção das extensões no sistema é ainda mais importante no caso de autorização não-privilegiada, pois as extensões, a priori, não são confiáveis. Desta forma, as extensões possuem mais chances de comprometer a corretude do sistema.

Autorização *privilegiada* é a forma de autorização que permite que apenas um determinado conjunto de usuários, ditos privilegiados, possam instalar extensões no sistema. Nessa forma de autorização o sistema valida se o usuário pertence ao conjunto de usuários que pode instalar extensões e, em caso positivo, libera a instalação. Normalmente, o conjunto de usuários que pode instalar extensões é definido por um grupo no sistema operacional. Além disso, é possível definir diferentes grupos com diferentes níveis de privilégio. Por exemplo, usuários de um grupo em particular podem deter privilégio de instalação de extensões no sistema de arquivos virtual e não no escalonador de processos. Contudo, o caso mais comum é termos apenas um usuário ou grupo de usuários com privilégio de instalar extensões irrestritamente no sistema como um todo, os chamados superusuários (e.g., *root* nos sistemas tipo-UNIX). Outra forma de se prover autorização da extensão da instalação

das extensões é dar privilegio apenas para o distribuidor do sistema instalar extensões, o que é mais freqüente em sistemas proprietários. A verificação da procedência da extensão pode ser feita, por exemplo, utilizando certificados digitais. Alguns exemplos de autorização privilegiada são sistemas que se baseiam em módulos carregáveis de *kernel*, como Linux, NetBSD e Solaris; esses sistemas permitem apenas que os usuários do grupo *root* instalem extensões no sistema. Garantir a proteção das extensões é geralmente tido como menos importante no caso de autorização privilegiada, pois as extensões, a priori, são confiáveis. Desta forma, as extensões possuem (em tese) menor chance de comprometer a corretude do sistema. Portanto, autorização privilegiada é geralmente associada com proteção por "boa vontade".

Duração

Duração é a característica que determina quanto tempo as extensões permanecem instaladas no sistema. Extensões podem ser classificadas de quatro formas quanto a duração, de acordo com a condição que esteja atrelada: aplicação, recurso, *kernel*, permanente.

Duração condicionada a uma aplicação significa que a extensão permanecerá instalada no sistema até que a aplicação a qual a extensão está atrelada encerre a sua execução.

Duração condicionada a um recurso significa que a extensão permanecerá instalada no sistema enquanto o recurso existir. Esse recurso pode ser, por exemplo, um arquivo ou um dispositivo.

Duração condicionada ao *kernel* significa que a extensão permanecerá instalada no sistema até que o *kernel* encerre a sua execução, ou seja, até que o sistema seja reiniciado.

Duração permanente significa que a extensão permanecerá instalada no sistema até que ela seja explicitamente removida.

Granularidade

Granularidade é a característica que determina a unidade de modificação de uma extensão. A granularidade do sistema pode ser classificada de três formas: modular, procedural e procedural limitada.

Granularidade modular é uma forma de granularidade grosseira (*coarse grain*). Isto significa que as extensões podem substituir e adicionar apenas módulos ou subsistemas inteiros no sistema operacional. Alguns exemplos de granularidade modular são módulos carregáveis de *kernel* em sistemas monolíticos e processos servidores em sistemas *microkernel*.

Granularidade procedural é uma forma de granularidade fina (*fine grain*). Isto significa que as extensões podem substituir e adicionar qualquer procedimento individual no sistema operacional. Alguns exemplos de granularidade procedural são as bibliotecas de sistema operacional do Exokernel, responsáveis por implementar abstrações tradicionalmente providas pelo sistema operacional (2).

Granularidade procedural limitada também é uma forma de granularidade fina, contudo, limitada. Isto significa que as extensões podem substituir e adicionar apenas alguns procedimentos individuais no sistema operacional. Alguns exemplos de granularidade procedural limitada são sistemas com proteção de extensões de *kernel* baseada em linguagem como SPIN, VINO e Exokernel.

Arbitragem

Arbitragem é a característica que trata a resolução de conflitos entre extensões. Extensões podem conflitar de três maneiras. Podem conflitar por requisitarem o mesmo recurso físico (e.g., o mesmo endereço de memória), o que é chamado de *interferência*. Podem conflitar por requisitarem mais frações de um mesmo recurso do que encontra-se disponível no momento (e.g., alocação de memória), o que é chamado de *contenção de recurso*. Por fim, extensões podem conflitar por requisitarem diferentes políticas globais para o sistema (e.g., escalonamento de processos), o que é chamado de *contenção de política*.

Existem três abordagens básicas para tratar essas formas de conflito. A abordagem mais simples é desabilitar os conflitos, o que é chamado de *proibição*. Nesta abordagem, as requisições à um recurso simplesmente falham quando existe um conflito. Isto pode ser implementado, por exemplo, através de uma autoridade central que arbitra os pedidos de recursos (e.g., alocador de memória).

Outra abordagem é decompor o processo de decisão em decisões globais e locais, o que é chamado de *decisão separada* Nesta abordagem, decisões globais são tomadas pelo sistema operacional ou por apenas uma extensão específica, designada para atuar com exclusividade sobre o sistema; decisões locais podem ser tomadas por extensões designadas para agir sobre determinado domínio (e.g., arquivo, processo). Desta forma, os conflitos são evitados, pois cada extensão atua exclusivamente sobre determinado escopo. Por exemplo, podemos ter uma extensão responsável pela política global de escalonamento do sistema operacional e extensões responsáveis pela política de escalonamento de conjuntos determinados de processos ou *threads*; assim, o escalonador global recorreria aos escalonadores locais para decidir qual seria o próximo processo a ser executado dentre um determinado conjunto de processos.

A última forma de tratar os conflitos entre extensões é a *multiplexação* do acesso a dado recurso no tempo. Nesta abordagem, é dada uma fatia de tempo para cada extensão utilizar dado recurso. Por exemplo, duas extensões que precisam manipular a tabela de interrupções do sistema, podem fazê-lo em tempos diferentes. Isto pode ser implementado através do agendamento de solicitações ou exclusão mútua a determinado recurso, por exemplo.

Programabilidade

Programabilidade é a característica que propomos para determinar como as extensões podem ser programadas, ou seja, o nível de abstração das linguagens e das APIs. Classificamos a programabilidades de quatro formas: linguagem de sistema, linguagem de sistema limitada, linguagem de extensão e linguagem de domínio específico.

Linguagem de sistema é a mesma linguagem em que o sistema foi desenvolvido sem restrições e com todas as APIs disponíveis. Este é o caso, por exemplo, de módulos carregáveis de *kernel*.

Linguagem de sistema limitada é a mesma linguagem em que o sistema foi desenvolvido com algumas restrições, como verificação de código ou APIs limitadas. Este é o caso, por exemplo, de alguns sistemas com proteção baseada em linguagem, como SPIN, VINO e Singularity. Nesses sistemas temos verificação de código das extensões e/ou um subconjunto da linguagem de programação e APIs disponíveis para as extensões.

Linguagem de extensão é uma linguagem específica desenvolvida para escrever as extensões do sistema. Esse tipo de linguagem apresenta necessariamente um nível de abstração mais alto do que a linguagem de programação do sistema e as respectivas APIs dela. Este é o caso, por exemplo, do μChoices (17) que usa a linguagem de script Tcl para a escrita das extensões.

Linguagem de domínio específico é uma linguagem desenvolvida com um propósito específico para atender a uma determinada classe de problemas. Normalmente, linguagens desse tipo não são Turing-completas e são bastante limitadas. Este é o caso, por exemplo, do Exokernel, que possui linguagens de domínio específico para filtros de pacote e criação de sistemas de arquivo.

2.3 Linguagens de Script

Linguagens de script são geralmente definidas como linguagens dinamicamente tipadas, de altíssimo nível (muitas instruções de máquina por comando da linguagem), interpretadas e voltadas para a "colagem" de programas. Chamamos de "colagem" a integração de diferentes partes ou componentes de

um programa através de um script que define o comportamento final do programa (7).

Alguns exemplos de linguagem de script são Lua, Tcl, Perl, Python e Ruby. As linguagens de script têm raízes nas linguagens de controle de *jobs* usadas em sistemas de processamento de lote, como a JCL (*Job Control Language*) usada no OS/360 da IBM, para descrever o sequenciamento da execução de tarefas (*jobs*). As linguagens de controle de *jobs*, por sua vez, evoluíram para linguagens mais sofisticadas, contando com controles de fluxo e até estrutura de dados avançadas. Estas linguagens são chamadas de linguagens de *shell scripting*, pois são utilizadas por interpretadores de linha de comando de sistemas operacionais (*shell*), como sh, ksh, csh e bash. Essas raízes em linguagens de controle de *jobs* e em linguagens de *shell scripting* demonstram a forte relação entre linguagens de script e sistemas operacionais. Algumas linguagens de script, como Tcl e Perl, apresentam clara influência de linguagens de *shell scripting* e forte integração com o sistema operacional. Além das raízes em sistemas operacionais, algumas linguagens (como Lua) apresentam inspiração em linguagens descritoras de dados ou parâmetros (como bibtex).

Segundo Ousterhout (7), linguagens de script contrastam com outro tipo de linguagem, as linguagens ditas de programação de sistema. Linguagens de programação de sistema são geralmente definidas como linguagens estaticamente tipadas, de alto nível e compiladas. Alguns exemplos de linguagem de programação de sistema são C, C++, Pascal, Modula-3 e Ada.

Embora as linguagens de script contrastem com as linguagens de programação de sistemas, elas não são concorrentes e sim complementares, pois o propósito de uma linguagem de script não é a substituição de uma linguagem de programação de sistemas e sim ser utilizada em conjunto. A idéia central do desenvolvimento de programas utilizando scripts é o uso de diferentes ferramentas para diferente tarefas, ou seja, o uso da ferramenta de programação mais adequada para cada tipo de tarefa. Assim, separa-se o programa em duas partes, uma responsável pelos seus componentes básicos e outra responsável pelo seu comportamento final. A parte responsável pelos componentes básicos do programa é tipicamente escrita em uma linguagem de programação de sistema. A parte responsável pelo comportamento final do programa é tipicamente escrita em uma linguagem de script.

Linguagens de script mostram-se mais adequadas para integração de programas pela sua natureza dinâmica e de altíssimo nível, o que se reflete em maior agilidade e facilidade de desenvolvimento. Em grande parte dos casos, se reduz bastante o tempo de desenvolvimento e a quantidade de linhas de código do programa. Entretanto, também em grande parte dos casos, diminui-se o

desempenho do programa, pois linguagens interpretadas e de mais alto nível são comumente mais lentas que linguagens de mais baixo nível e compiladas. Podemos tecer a mesma comparação entre linguagens compiladas e linguagens de *assembly*; linguagens compiladas são geralmente mais lentas.

Linguagens de programação de sistema mostram-se mais adequadas para o desenvolvimento de algoritmos e estruturas de dados complexos, pois linguagens com tipagem estática facilitam o gerenciamento de código e a detecção de erros pelo compilador. Linguagens de programação de sistema também se mostram mais adequadas para tarefas que exigem maior proximidade com o *hardware*, por serem menos abstratas que as linguagens de script.

Essa abordagem, de utilizar linguagens de script em conjunto com linguagens de programação de sistema, também aumenta a flexibilidade do programa, pela natureza interpretada das linguagens de script.

2.3.1 Estender vs. Embutir

Existem basicamente duas abordagens para se utilizar uma linguagem de script em conjunto com uma linguagem de programação de sistema para o desenvolvimento de um programa: estender ou embutir o interpretador da linguagem de script (24, 25).

Na primeira abordagem, *estender o interpretador*, como o próprio nome da abordagem sugere, o programa desenvolvido estende a linguagem de script, adicionando nela novas funcionalidades. O núcleo do programa desenvolvido, tipicamente, é implementado sob a forma de uma ou mais bibliotecas de extensão e é escrito em uma linguagem de programação de sistema (normalmente a mesma linguagem utilizada na implementação do interpretador). O interpretador, nesse caso, detém o fluxo de execução principal do programa (e.g., função `main` na linguagem C) e é normalmente utilizado apenas para executar o script. Desta forma, o script, detém o fluxo de execução efetivo (ou lógico) do programa, uma vez que o interpretador é apenas responsável por executá-lo. O script, então, é responsável pelo comportamento final do programa e eventualmente invoca o núcleo do programa (sob a forma de bibliotecas).

Nessa abordagem, o programa completo é composto pelo interpretador da liguagem de script (programa principal na linguagem de programação de sistema), pelo núcleo do programa (bibliotecas de extensão do interpretador) e pelo script (programa principal lógico). Considere, por exemplo, que desejamos escrever um aplicativo científico para cálculo numérico utilizando essa abordagem. Então, tipicamente, implementaríamos bibliotecas de extensão para o interpretador contendo diferentes métodos de cálculo numérico. Este seria o

núcleo do nosso programa. Implementaríamos também um script responsável pelo comportamento final do programa. Este script faria a interface com o usuário e realizaria as operações desejadas por ele, invocando as bibliotecas (núcleo do programa).

Na segunda abordagem, *embutir o interpretador*, como o próprio nome da abordagem sugere novamente, o interpretador é adicionado (embutido) no programa desenvolvido. Isto é feito tipicamente utilizando o interpretador como uma biblioteca de extensão para o núcleo do programa desenvolvido. Nesse caso, o núcleo normalmente detém o fluxo de execução principal do programa e utiliza o interpretador para invocar o script. O script também é utilizado para determinar o comportamento final do programa. Contudo, ao invés do script deter o fluxo de execução principal do programa, ele é chamado pelo núcleo do programa. As chamadas ao script são feitas tipicamente através de ligações (*bidings*) às funções implementadas no script. Essas ligações são acessadas pelo núcleo através da biblioteca que implementa o interpretador.

Nessa abordagem, o programa completo nesse caso também é composto pelo interpretador da liguagem de script (biblioteca de extensão do núcleo), pelo núcleo do programa (programa principal) e pelo script (funções chamadas pelo núcleo). Considere novamente que desejamos escrever o aplicativo científico para cálculo numérico que mencionamos acima, mas, agora, utilizando essa segunda abordagem. Então, tipicamente, implementaríamos os diferentes métodos de cálculo numérico também no núcleo do programa. Além disso, criaríamos ligações no núcleo para possibilitar as chamadas às funções implementadas no script. O script continuaria implementando a interface com o usuário e as operações requiridas por ele. Porém, nesse caso, o núcleo seria responsável por invocar o script e não o contrário.

Embora não seja necessário, é usual exportar funções para o script através de extensão do interpretador também na abordagem de *estender o interpretador*. Por exemplo, os métodos de cálculo numérico poderiam ser exportados para o script através de bibliotecas de extensão ou ser passados para o script através de parâmetros de função.

2.3.2 Por Que Lua?

Lua é uma linguagem de script dita extensível e de extensão, projetada para, ao mesmo tempo, customizar e ser customizada por aplicações (8, 26). Diferentemente da maioria das linguagens de script, Lua foi projetada especificamente como uma linguagem embutível, isto é, ela é implementada como uma biblioteca normal de C e possui uma API bem definida (27).

Utilizar a estratégia de *scripting* para desenvolver ou estender um *kernel* de sistema operacional é diferente de utilizá-la em uma aplicação executada no espaço de usuário, pois *kernels* de sistema operacional são suscetíveis a um conjunto particular de restrições. Todavia, além de ser, ao mesmo tempo, facilmente embutível e estendível (ou seja, suporta os dois modos de *scripting*), Lua provê algumas outras facilidades que fazem dela uma escolha bastante adequada para um ambiente de *scripting* de *kernel* de sistema operacional.

Kernels de sistema operacional são tipicamente escritos utilizando um subconjunto limitado de funcionalidades da linguagem de programação C. Não existe suporte para tipos de ponto flutuante, pois as trocas de contexto da unidade de ponto flutuante são demasiadamente custosas. Também não existe suporte para a biblioteca padrão de C completa; em vez disso, existe apenas um conjunto limitado de funções disponível. Por exemplo, o código do *kernel* não pode usar as tradicionais funções `free` e `malloc`, pois elas dependem do *kernel* para serem implementadas.

De acordo com o padrão ISO C (28), um ambiente *freestanding* é aquele que não faz uso de nenhum benefício provido pelo sistema operacional. Programas que são compatíveis com essa determinação de ambiente têm apenas um conjunto limitado de cabeçalhos padrões C disponíveis: `float.h`, `iso646.h`, `limits.h`, `stdarg.h`, `stdbool.h`, `stddef.h` e `stdint.h`. Idealmente, *kernels* de sistema operacional e suas extensões devem ser compatíveis com a especificação *freestanding*. Além disso, como sistemas operacionais de propósito geral são projetados para executar sobre várias plataformas de *hardware* diferentes, um interpretador de linguagem de script embutido não deve ter código dependente de plataforma, como por exemplo código dependente de *endianess*.

Como portabilidade é um dos principais objetivos de projeto de Lua, o seu núcleo é praticamente compatível com a especificação *freestanding*, dependendo apenas de alguns poucos cabeçalhos adicionais. Essas dependências adicionais, contudo, podem ser facilmente rastreadas no código fonte de Lua, pois elas estão todas confinadas a um único arquivo de cabeçalho de configuração. O código dependente de ponto flutuante está igualmente confinado a esse arquivo de cabeçalho de configuração. Além disso, todo o código dependente de sistema operacional é localizado fora do núcleo de Lua, em bibliotecas externas. O núcleo de Lua, por exemplo, não é ligado às funções de alocação de memória de C; ele aloca memória chamando uma função que é passada como um argumento quando um novo estado é criado. Lua também é totalmente escrita em puro ISO C (28) e não é dependente de plataforma de *hardware*.

Outra restrição de um *kernel* de sistema operacional é o tamanho. Um kernel de sistema operacional permanece carregado na memória a partir do

momento que o sistema é inicializado e até que ele seja desligado. Portanto, o tamanho do *kernel* é uma questão bastante relevante. Imagens binárias de *kernels* para a arquitetura IA32 geralmente possuem menos que 15 MB (distribuições comuns de Linux possuem cerca de 3 MB e NetBSD possui cerca de 10 MB)[1]. Assim, da mesma forma que o próprio *kernel*, um interpretador embutido nele também deve ser suficientemente pequeno e eficiente para tratar essa restrição.

Comparado com outras linguagens de script, Lua possui pequena ocupação de memória. A versão 5.1.4 do interpretador *standalone* de Lua, em conjunto com todas as bibliotecas padrão de Lua, tem 258 KB no Ubuntu 10.10; outras linguagens de script, como Python e Perl, possuem o tamanho de alguns megabytes[2], a mesma ordem de grandeza de um *kernel* completo de sistema operacional.

Finalmente, como um *kernel* de sistema operacional possui acesso irrestrito ao *hardware* completo, restrições especiais devem ser atribuídas as extensões de *kernel* para evitar que danos ou acessos indesejados aos recursos do sistema. Assim, a linguagem embutida deve prover alguma forma de isolar o código das extensões e restringir o acesso deles ao ambiente de *kernel*.

Lua provê suporte de programação para garantir restrições de acesso aos scripts. Assim como a maioria das linguagens de script, Lua possui gerenciamento automático de memória, o que previne scripts de manipular diretamente a memória usando ponteiros. Lua também suporta múltiplas instâncias de estados do interpretador. Na realidade, a implementação do interpretador Lua não possui nenhuma variável global, todos os dados são mantidos em uma estrutura de dados, chamada de estado Lua (*Lua state*). Assim, para se obter o isolamento completo entre estados de execução, basta-se criar diferentes estados Lua utilizando a API C de Lua. Além disso, como podemos ter diferentes conjuntos de bibliotecas disponíveis para diferentes estados, é possível criar diferentes domínios de proteção para diferentes níveis de privilégio.

Além disso, é importante observar que todas as restrições que apresentamos não são exclusivas de *kernels* de sistema operacional; pelo contrário, vários outros ambientes de programação também apresentam algumas ou todas essas restrições, como por exemplo *sistemas embutidos*, jogos e até mesmo alguns aplicativos de usuário.

No próximo capítulo, apresentaremos a nossa abordagem para desenvolvimento de sistemas operacionais que utiliza os conceitos de sistema operacionais

[1]No Ubuntu 10.10, a imagem do *kernel* genérico possui 3,9 MB. No NetBSD 5.1, a imagem do *kernel* genérico para a arquitetura IA32 possui 11 MB.

[2]No Ubuntu 10.10, Python 2.6.5 e Perl 5.10.1 têm, respectivamente, 2,21 e 1,17 MB.

extensíveis e linguagens de script: *sistemas operacionais scriptáveis*.