Pontifícia Universidade Católica
do Rio de Janeiro

Lourival Vieira Neto

Lunatik: Scripting de Kernel de Sistema Operacional com Lua

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC–Rio

Orientador: Prof. Roberto Ierusalimschy

Rio de Janeiro
Abril de 2011

Pontifícia Universidade Católica
do Rio de Janeiro

Lourival Vieira Neto

Lunatik: Scripting de Kernel de Sistema Operacional com Lua

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática — PUC–Rio

Prof. Noemi de La Rocque Rodriguez
Departamento de Informática — PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 12 de Abril de 2011

Lourival Vieira Neto

Graduou-se em Engenharia de Computação pela Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), em Dezembro de 2008.

Ficha Catalográfica

Vieira Neto, Lourival

Lunatik: Scripting de Kernel de Sistema Operacional com Lua / Lourival Vieira Neto; orientador: Roberto Ierusalimschy. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2011.

v., 69 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Sistemas Operacionais Extensíveis;. 3. Scripting;. 4. Kernel;. 5. Lua.. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

À minha família, aos meus amigos e aos meus professores.

Resumo

Vieira Neto, Lourival; Ierusalimschy, Roberto. **Lunatik: Scripting de Kernel de Sistema Operacional com Lua**. Rio de Janeiro, 2011. 69p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Existe uma abordagem de projeto para aumentar a flexibilidade de sistemas operacionais, chamada sistema operacional extensível, que sustenta que sistemas operacionais devem permitir extensões para poderem atender a novos requisitos. Existe também uma abordagem de projetos no desenvolvimento de aplicações que sustenta que sistemas complexos devem permitir que usuários escrevam scripts para que eles possam tomar as suas próprias decisões de configuração em tempo de execução. Seguindo estas duas abordagens de projeto, nós construímos uma infra-estrutura que possibilita que usuários carreguem e executem dinamicamente scripts Lua dentro de kernels de sistema operacional, aumentando a flexibilidade deles. Nesta dissertação, nós apresentamos Lunatik, a nossa infra-estrutura para scripting de kernel baseada em Lua, e mostramos um cenário de uso real no escalonamento dinâmico da freqüência e voltagem de CPU. Lunatik está implementado atualmente tanto para NetBSD quanto para Linux.

Palavras–chave

Sistemas Operacionais Extensíveis; Scripting; Kernel; Lua.

Abstract

Vieira Neto, Lourival; Ierusalimschy, Roberto (Adviser). **Operating System Kernel Scripting with Lua**. Rio de Janeiro, 2011. 69p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

There is a design approach to improve operating system flexibility, called extensible operating system, that supports that operating systems must allow extensions in order to meet new requirements. There is also a design approach in application development that supports that complex systems should allow users to write scripts in order to let them make their own configuration decisions at run-time. Following these two design approaches, we have built an infrastructure that allows users to dynamically load and run Lua scripts into operating system kernels, improving their flexibility. In this thesis we present Lunatik, our scripting subsystem based on Lua, and show a real usage scenario in dynamically scaling CPU frequency and voltage. Lunatik is currently implemented both for NetBSD and Linux.

Keywords

Extensible Operating Systems; Scripting; Kernel; Lua.

Sumário

Lista de figuras

Lista de tabelas

The lunatic is on the grass
The lunatics are on the grass
Remembering games
And daisy chains and laughs
Got to keep the loonies on the path

The lunatic is in the hall
The lunatics are in my hall
The paper holds their folded faces to the floor
And every day the paper boy brings more

And if the dam breaks open many years too soon
And if there is no room upon the hill
And if your head explodes with dark forebodings too
I'll see you on the dark side of the moon

The lunatic is in my head
The lunatic is in my head
You raise the blade
You make the change
And you rearrange me 'till I'm sane
You lock the door
And throw away the key
And there's someone in my head, but it's not me

And if the cloud bursts thunder in your ear
You shout and no one seems to hear
And if the band you're in starts playing different tunes
I'll see you on the dark side of the moon

Roger Waters, *Brain Damage.*

1
Introdução

Sistema operacional extensível é uma abordagem de projeto de sistemas operacionais que foi originada no final da década de 1960 (1) e tem sido bastante explorada a partir da década de 1990 até a atualidade (2, 3, 4, 5). Nessa abordagem, é sustentado que podemos aumentar a flexibilidade, confiabilidade e desempenho de um sistema operacional através da sua extensibilidade. Essa abordagem se baseia no princípio que sistemas operacionais precisam permitir o uso de extensões para atender a mudanças nos seus requisitos iniciais de projeto (1). Os sistemas operacionais extensíveis focam na idéia-chave de que sistemas operacionais de uma forma geral não são capazes de antever os requisitos de todas as suas aplicações, portanto eles devem permitir a adaptação do seu funcionamento para poder atender a novos requisitos (6).

Em outro contexto, o desenvolvimento de aplicações, existe uma importante tendência em dividir sistemas complexos em duas partes — *núcleo* e *configuração* — e em combinar linguagens de programação de sistema e linguagens de script para o desenvolvimento dessas partes (7, 8). A linguagem de programação de sistema é utilizada para escrever os componentes que formam o núcleo do sistema; e a linguagem de script é utilizada para escrever a configuração, ou seja, definir o comportamento final do sistema, ligando os componentes do núcleo (7, 8). Linguagens de programação de sistema, como por exemplo C e C++, são linguagens de mais baixo nível e geralmente estaticamente tipadas e compiladas. Linguagens de script, como por exemplo Lua, Tcl, Python e Perl, são linguagens de mais alto nível e geralmente dinamicamente tipadas e interpretadas. Essa abordagem de desenvolvimento de aplicativos é comumente chamada de *scripting*. O uso desta abordagem traz vantagens significativas ao desenvolvimento de aplicativos. Primeiro, ela aumenta a produtividade de desenvolvimento, oferecendo um ambiente mais ágil pelo uso de uma linguagem de mais alto nível e interpretada (7). Segundo, ela aumenta a flexibilidade do desenvolvimento, pois oferece uma linguagem para a extensão do aplicativo em tempo de execução, em vez da simples customização através da configuração de parâmetros (8).

Neste trabalho, nós propomos a junção dessas duas abordagens de de-

senvolvimento — *sistema operacional extensível* e *scripting* — para possibilitar o uso de linguagens de script no desenvolvimento de sistemas operacionais. Nós chamamos essa nossa proposta de abordagem de *sistema operacional scriptável*. De forma semelhante ao *scripting* de aplicativos, um sistema operacional scriptável utiliza scripts para ligar seus componentes e definir o seu comportamento final. De forma semelhante a um sistema operacional extensível, um sistema operacional scriptável possibilita a extensão do seu funcionamento, contudo, utilizando uma linguagem de script como linguagem de desenvolvimento das extensões.

Baseados na idéia de sistema operacional scriptável, desenvolvemos um protótipo para possibilitar o *scripting* de *kernel* de sistema operacional utilizando Lua, ou seja, para possibilitar o *scripting* da principal parte de um sistema operacional. Desenvolvemos esse protótipo, chamado *Lunatik*, para NetBSD e Linux. O desenvolvimento de Lunatik foi iniciado no nosso trabalho anterior com o objetivo apenas de executar scripts Lua dentro do *kernel* Linux (9). Desta forma, a primeira versão de Lunatik consistia basicamente em um interpretador Lua embutido no *kernel* Linux. A versão de Lunatik que apresentamos nessa dissertação é uma infra-estrutura de *scripting* que possui, além de um interpretador Lua embutido em *kernel*, uma interface de programação para desenvolvedores de *kernel* e uma interface de usuário para a carga de scripts no *kernel*. Entretanto, Lunatik não apresenta um sistema operacional scriptável em si, e sim um conjunto de ferramentas que possibilitam a adaptação de *kernels* com objetivo de torná-los "scriptáveis".

O resto desta dissertação é organizada como segue. O capítulo 2 apresenta os conceitos básicos que serão utilizados no restante desta dissertação. A seção 2.1 apresenta a definição para o conceito de sistema operacional. A seção 2.2 apresenta uma discussão sobre sistemas operacionais extensíveis e seus requisitos e características de projeto e implementação; A seção 2.3 apresenta uma discussão sobre linguagens de script e seu uso e motiva a escolha de Lua como linguagem para *scripting* de *kernels* de sistema operacional. O capítulo 3 descreve a nossa proposta de sistema operacional scriptável, apresenta uma discussão sobre suas principais características, alguns casos de uso e trabalhos relacionados. O capítulo 4 apresenta Lunatik, o nosso protótipo para *scripting* de *kernel* baseado em Lua; descreve o seu projeto e implementação; e apresenta experimentos de *scripting* de *kernel*. O primeiro experimento foi aplicado ao subsistema Cpufreq do Linux, que é responsável pelo controle de freqüência e voltagem de CPU (*central processing unit*) no Linux. O segundo experimento foi aplicado à composição de chamadas de sistema no NetBSD. Finalmente, o capítulo 5 apresenta as nossas conclusões.

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*.

3
Sistema Operacional Scriptável

Sistema operacional scriptável é a nossa proposta de modelo de projeto de sistema operacional com o objetivo de aumentar a sua flexibilidade e facilidade de desenvolvimento, através da junção de extensibilidade de sistema operacional e linguagens de script. A idéia central dessa abordagem é permitir o desenvolvimento e a extensão do sistema operacional de forma semelhante com que aplicativos de usuário são desenvolvidos e estendidos utilizando linguagens de script, tanto estendendo quanto embarcando o interpretador da linguagem. Desta forma, um sistema operacional scriptável é também um sistema operacional extensível que utiliza uma linguagem de script para escrever as extensões. A seguir descreveremos a caracterização de um sistema operacional scriptável baseados nos mesmos pontos de projeto e implementação descritos para sistemas operacionais extensíveis.

3.1
Projeto e Implementação

Nesta seção descreveremos as principais características de projeto e implementação de sistemas operacionais scriptáveis.

Mutabilidade

Sistema operacionais scriptáveis assumem dois modos de mutabilidade. Linguagens de script muitas vezes são utilizadas como simples linguagens para a configuração de um programa através do preenchimento de parâmetros. Isto faz com que um sistema operacional scriptável seja ao mesmo tempo um sistema reconfigurável e extensível, sob o ponto de vista da mutabilidade, pois ele possibilita ao mesmo tempo a adição de novo código ao sistema e a simples reconfiguração de parâmetros. Desta forma, a linguagem de script poderia substituir ou se integrar a mecanismos de reconfiguração de parâmetros como *sysfs* e *sysctl*.

Localização

Scripts podem ser usados tanto para estender o próprio *kernel* quanto para estender porções do sistema operacional executadas fora do *kernel*. Contudo, acomodar extensões no *kernel* agrega vantagens em flexibilidade e desempenho, como argumentado na seção 2.2.1, o que chamamos de *scripting* de *kernel* de sistema operacional e este é o foco da nossa implementação.

Proteção

A proteção dos scripts é feita por *software*, utilizando o isolamento provido pelo interpretador da linguagem (e.g., gerenciamento automático de memória). Parte da proteção pode ser feita também por *hardware*, como por exemplo preempção baseada em *threads* de *kernel* para garantir que o sistema não terá problemas de vivacidade (*liveness*). Entretanto, existem partes do *kernel* que não podem ser tratadas dentro de *threads* (e.g., tratadores de interrupção); neste caso, utilizamos a autorização para garantir que apenas scripts confiáveis possam comprometer a vivacidade do sistema. Outro recurso que pode ser utilizado para garantir a corretude é o oferecimento de interfaces sem estado (*stateless*) para garantir que scripts não violem os protocolos de uso das interfaces, seja acidentalmente ou propositalmente.

Duração

A duração dos scripts é condicionada à duração da instância do interpretador da linguagem onde o script é executado. O interpretador da linguagem de script, por sua vez, pode ser instanciado por aplicação, recurso ou *kernel*. Além disso, podemos ter o estado das instâncias salvo entre execuções do *kernel*. Assim, scripts podem ter a duração condicionada a aplicação, recurso, *kernel* ou permanente.

Granularidade

Sistemas operacionais scriptáveis oferecem granularidade procedural limitada. Esse tipo de sistema permite a modificação ou adição de código no sistema operacional através de ligações entre as extensões, implementadas em linguagem de script, e o sistema operacional, implementado em linguagem de sistema. Através dessas ligações é possível definir funções para serem chamadas pelo sistema operacional no lugar das implementações originais.

Arbitragem

Podemos misturar os três tipos de arbitragem (proibição, decisão separada e multiplexação) em um sistema operacional scriptável. A proibição e multiplexação podem ser implementadas através da API exportada para os scripts, utilizando mecanismos de sincronização nas interfaces exportadas. A segmentação pode ser implementada utilizando várias instâncias ou estados do interpretador da linguagem, diferenciando estados globais e locais.

Programabilidade

A programabilidade em um sistema scriptável é dada obviamente por uma linguagem de extensão, mais precisamente, uma linguagem de script. Além disso, a linguagem de script pode servir como uma ferramenta para a criação de linguagens de domínio específico para diferentes domínios de extensão de um sistema operacional. Desta forma, a programabilidade de um sistema scriptável pode ser considerada a combinação entre a programabilidade dada por uma linguagem de extensão e uma linguagem de domínio específico.

Autorização

A autorização de um sistema operacional scriptável pode ser feita combinando-se diversos níveis de privilégios, segmentando a autorização ao uso de scripts dentro do kernel a diversos domínios de segurança separados por estados diferentes do interpretador. Desta forma, temos um controle mais granular da autorização, permitindo que grupos de usuários específicos tenham acesso a estados específicos do interpretador.

3.2 Casos de Uso

Elaboramos alguns casos de uso que esperamos tratar com *scripting* de *kernel* de sistema operacional, tanto casos embutir quanto de estender o interpretador Lua. Nesta seção, descreveremos alguns deles.

Escalonador de Processos

A idéia desse caso de uso é possibilitar que usuários definam novos algoritmos para o controle global do escalonamento de processos ou que criem diferentes politicas de escalonamento para conjuntos independentes de processos ou *threads*, em tempo de execução. Isto poderia, por exemplo, auxiliar na implementação de qualidade de serviço *(QoS)* em processos. Esse caso utiliza a abordagem de embutir o interpretador.

Filtro de Pacotes

O propósito desse caso é possibilitar a criação de regras mais elaboradas para o processamento e filtragem do tráfego de pacotes de rede, em vez de usar simples tabelas de regras. Por exemplo, um usuário poderia criar sua própria implementação para inspeção de pacotes ou para tradução de endereços de rede *(NAT)*. Esse caso utiliza a abordagem de embutir o interpretador.

Gerenciamento de Energia

O propósito desse caso é possibilitar que usuários definam os seus próprios métodos para o gerenciamento do consumo de energia no sistema. Por exemplo, usuários poderiam definir algoritmos para escalonar a freqüência e voltagem da unidade central de processamento *(CPU)* para poupar energia. Nós exploramos esta idéia em mais detalhes na seção 4.3. Esse caso utiliza a abordagem de embutir o interpretador.

Drivers de Dispositivo

Existem algumas iniciativas de utilização de linguagens seguras e de domínio específico para escrever *drivers* de dispositivos, como NDL (29). A idéia desse caso é utilizar uma linguagem de scripting tanto para fazer a configuração dos drivers (normalmente utilizando tabelas de parâmetros) quanto para implementar as funções necessárias para o funcionamento adequado do dispositivo e criar drivers especializados por aplicação (e.g., comprimir dados durante o acesso ao disco). Esse caso utiliza tanto a abordagem de embutir e quanto a de estender o interpretador.

Banco de Dados

O propósito desse caso é utilizar scripts para, por exemplo, filtrar as buscas em arquivos de *backend* em sistemas de gerenciamento de banco de dados, evitando realizar sucessivas chamadas de sistema para processar o filtro em nível de usuário. Esse caso utiliza a abordagem de estender o interpretador, pois o sistema de gerenciamento de banco de dados faria as chamadas ao script (de forma semelhante a uma chamada de sistemas).

Servidor Web

O propósito desse caso é utilizar scripts para implementar a pilha ou parte da pilha de protocolo HTTP de um servidor *web* dentro do kernel, também poupando diversas trocas de contexto. Esse caso utiliza a abordagem de estender o interpretador, pois o aplicativo faria explicitamente as chamadas

ao script (também de forma semelhante a uma chamada de sistemas). Isso poderia ser feito, por exemplo, através do *scripting* de BSD Sockets.

3.3 Trabalhos Relacionados

Nesta seção descrevemos alguns trabalhos relacionados à nossa proposta de sistema operacional scriptável e *scripting* de *kernel*. Embora existam muitas abordagens para sistemas operacionais extensíveis e usos de recursos de linguagem de programação no desenvolvimento de sistemas operacionais, abordaremos nesta seção apenas trabalhos que utilizam formas semelhantes à da nossa proposta, ou seja, utilizam interpretadores de linguagens com características de linguagens de script dentro do *kernel*.

Exokernel (13) é uma arquitetura de sistema operacional extensível desenvolvido pelo MIT que é baseada na idéia que o sistema operacional não deve prover abstrações, ou seja, não deve desempenhar o papel de máquina virtual. Em vez de prover abstrações, o sistema operacional deve ser exclusivamente responsável por gerenciar recursos de forma segura, atuando como um multiplexador do *hardware*. Para atingir esse objetivo, as implementações de Exokernel desenvolvidas pelo MIT utilizam linguagens de domínio específico para possibilitar que os usuários manipulem o disco e a rede. Os códigos escritos nessas linguagens são carregados pelos usuários e interpretados dentro do kernel. A principal diferença desta abordagem para a nossa é a utilização de linguagens de domínio específico e limitadas pelo Exokernel. O *scripting* de *kernel* poderia ser utilizado nesse caso para a construção das linguagens de domínio especifico, facilitando e homogenizando o desenvolvimento e manutenção de código.

Outro sistema que utiliza interpretador embutido no kernel é μChoices (17), um sistema operacional extensível baseado em microkernel. A idéia principal do μChoices é permitir uso de scripts escritos em Tcl para a criação de chamadas de sistemas customizadas para atender a necessidades específicas de aplicações. As chamadas de sistema customizadas são definidas através de scripts de usuário carregados no *kernel* do sistema operacional. Esses scripts definem as chamadas utilizando chamadas de sistema já existentes no sistema operacional, evitando assim o excesso de trocas de contexto para se realizar um lote de chamadas. A principal diferença desta abordagem para a nossa é que μChoices se baseia exclusivamente em prover *scripting* de *kernel* por embutir o interpretador Tcl no *kernel*. Enquanto Lunatik, a nossa implementação de *scripting* de *kernel*, provê *scripting* tanto por embutir quanto por estender o interpretador Lua.

Além desses exemplos de trabalhos relacionados, não é incomum o uso de linguagens de domínio específico dentro de *kernels* de sistema operacional, como por exemplo para a construção de filtro de pacotes ou acesso ao controle avançado de energia (*ACPI — Advanced Configuration and Power Interface*). Sistemas operacionais scriptáveis possibilitam a substituição dessas diversas linguagens de domínio específico por uma única linguagem de scripting, provendo um ambiente mais uniforme de desenvolvimento e mais fácil de se manter, no lugar de várias implementações de linguagens *ad-hoc*.

4
Lunatik

Lunatik é a nossa implementação de um protótipo para prover um sistema operacional scriptável. A abordagem escolhida não foi a de desenvolver um sistema operacional scriptável completamente do início. Em vez disso, optamos pelo desenvolvimento gradual. Implementamos o suporte a *scripting* de *kernel* para dois sistemas operacionais existentes, NetBSD e Linux. Lunatik é composto por um subsistema de *kernel* e um aplicativo em espaço de usuário. O subsistema de *kernel* oferece suporte para que os desenvolvedores de *kernel* adéqüem subsistemas para serem "scriptados", isto é, para que possam ser desenvolvidos ou estendidos utilizando scripts. O aplicativo de usuário oferece suporte a carga e execução dinâmica de scripts no *kernel* a partir do espaço de usuário. Assim, possibilitamos que o *kernel* possa ser desenvolvido e estendido utilizando a abordagem de *scripting*.

Lunatik suporta *scripting* de *kernel* de sistema operacional das duas formas mencionadas anteriormente: embutindo e estendendo o interpretador Lua. Embutir o interpretador Lua significa que os subsistemas de *kernel* se comportam como programas hospedeiros, invocando o interpretador Lua como um biblioteca para executar os scripts. Estender o interpretador Lua significa que os subsistemas de *kernel* se comportam como bibliotecas para os scripts, os quais detêm o controle de fluxo.

Para suportar o *scripting* de *kernel* embutindo o interpretador Lua, desenvolvedores de *kernel* precisam modificar os seus subsistemas para que estes invoquem o interpretador Lua para executar scripts. Através desses scripts, usuários podem adaptar o comportamento do sistema operacional às suas próprias demandas, definindo políticas e mecanismos apropriados às suas tarefas.

Para suportar o *scripting* de *kernel* estendendo o interpretador Lua, desenvolvedores de *kernel* precisam criar bibliotecas que exponham funções e estruturas de dados internas do *kernel* para os scripts. Assim, é fornecido acesso direto para o interior do *kernel* para os scripts, fazendo com que os scripts carregados no *kernel* possam evitar as interfaces tradicionais oferecidas para o nível de usuário, as quais normalmente envolvem sucessivas e custosas trocas

de contexto. Bibliotecas de extensão também podem ser utilizadas no caso de embutir o interpretador para prover acesso a algumas funções ou estruturas de dados internas do *kernel* para os scripts.

4.1 Visão Geral de Funcionamento

Nesta seção descreveremos o funcionamento básico de Lunatik através de exemplos dos dois casos de *scripting* de *kernel* (estendendo ou embutindo Lua). Esses exemplos são apenas ilustrativos com o objetivo didático de facilitar a explicação. Na seção 4.3, apresentaremos exemplos que foram de fato implementados em experimentos.

Scripting de Kernel Estendendo Lua

Suponha que o desenvolvedor do sistema de arquivos virtual (VFS) deseja prover acesso direto a este subsistema para os scripts, permitindo o scripting de kernel estendendo Lua. Por exemplo, um usuário poderia utilizar o script mostrado na figura 4.1 para estender o funcionamento do VFS com o seu próprio método de filtragem de leitura de um arquivo. A função `filter` lê um arquivo até o seu final e retorna para o espaço de usuário apenas o número de ocorrências de determinada string.

O primeiro passo de um subsistema que oferece *scripting*, como o VFS neste exemplo, é a criação de um novo estado Lua. Desta forma, é necessário que o desenvolvedor do subsistema modifique-o para que o subsistema crie um estado Lua. No passo seguinte, o subsistema deve carregar as bibliotecas que exportam as funções e estruturas de dados necessárias para que os scripts possam desempenhar a sua tarefa corretamente. É importante notar que a implementação dessas bibliotecas de extensão do interpretador Lua é responsabilidade dos desenvolvedores de *kernel*, de acordo com o que pretendem expor dos seus subsistemas para os scripts. Por exemplo, o script mostrado na figura 4.1 usa uma biblioteca que estende o interpretador Lua (`vfs`), exportando uma função para a leitura de arquivos (`vfs.read`) para os scripts.

O último passo que o subsistema tem que desempenhar é registrar o novo estado Lua no Lunatik para possibilitar que esse estado seja utilizado a partir do espaço de usuário. Após o registro ser efetuado, torna-se possível carregar e executar scripts nesse estado Lua a partir do espaço de usuário.

Nessa abordagem de scripting (estendendo Lua), scripts podem atuar fazendo uma "colagem", integrando chamadas de funções do kernel para fornecer uma nova "chamada de sistema" personalizada para o espaço de usuário. É importante observar que o que nós chamamos de *nova chamada*

```
function filter (f, s)
  local n = 0
  local rest = ""

  while true do
    local block = vfs.read(f, vfs.blocksize)
    if not block then break end

    block = rest .. block

    -- soma o número de ocorrências de s no bloco lido
    local m = select(2, string.gsub(block, s, s))

    n = m + n

    -- obtém o resto do bloco lido para testar no próximo
    local rest = string.sub(block, #block - #s + 2)
  end

  return n

end
```

Figura 4.1: Função filter em Lua

de sistema não é uma chamada de sistema tradicional; em vez disso, é uma chamada através de scripts utilizando Lunatik. Para chamar uma função definida em um script, é necessário apenas carregar e executar um outro script contendo uma chamada a essa função. A carga e execução de scripts é feita utilizando uma chamada de sistemas, que recebe uma *string* contendo o script e retorna uma outra *string* contendo o resultado da execução do script. Assim, pode-se obter valores Lua usando o comando `return` de Lua. Por exemplo, um usuário pode chamar a função `filter`, definida na figura 4.1, para obter o número de ocorrências da string `"Lua"` em um arquivo definido pelo descritor `fd` utilizando o seguinte script: `return filter(fd, "Lua")`. Neste caso, a função `filter` é executada e o seu valor de retorno é passado para o usuário através da chamada de sistema. Entraremos em mais detalhes quanto ao funcionamento da interface de usuário na seção 4.2.3.

Scripting de Kernel Embutindo Lua

Agora, suponha que o desenvolvedor do escalonador de processos do sistema operacional deseja permitir a definição de políticas de escalonamento através de funções definidas em scripts, permitindo, assim, scripting de kernel embutindo Lua. Por exemplo, um usuário poderia usar o script mostrado na figura 4.2 para estender o escalonador de processos do sistema operacional com uma implementação do algoritmo de Round-robin.

```
function scheduler.tick (run_queue, current_process)
  if (#run_queue > 0)
    -- desenfileira o próximo processo a ser executado
    next_process = scheduler.dequeue(run_queue)

    -- enfileira o processo atual
    scheduler.enqueue(run_queue, current_process)

    -- substitui o processo em execução
    scheduler.switchto(next_process)
  end
end
```

Figura 4.2: Algoritmo Round-robin em Lua

Nesta abordagem de scripting (embutindo Lua), scripts definem funções de *callback* para serem chamadas pelo subsistema do kernel. No nosso exemplo do escalonador de processos, esse subsistema chama o script através do nome fixado como `scheduler.tick`. Assim, o desenvolvedor precisa modificar o subsistema para chamar as funções que são implementadas pelos scripts em pontos apropriados do subsistema. A figura 4.2 mostra um exemplo de uma função que é chamada periodicamente pelo escalonador de processos para escolher o próximo processo a ser executado (`scheduler.tick`).

Assim como na abordagem anterior (estendendo Lua), o subsistema (o escalonador, neste exemplo) precisa criar um novo estado Lua, carregar as bibliotecas apropriadas para exportar as funções e estruturas de dados necessárias, e registrar o novo estado Lua no Lunatik. Por exemplo, a função mostrada na figura 4.2 usa uma nova biblioteca Lua (`scheduler`). Esta biblioteca acessa o interior do kernel para implementar uma função que substitui o processo em execução no sistema (`switchto`) e uma lista que representa a fila de processos que estão prontos para serem executados (`run_queue`).

4.2 Projeto e Implementação

Lunatik é um infra-estrutura que provê um ambiente de programação e execução para *scripting* de *kernel* de sistema operacional. Esse infra-estrutura foi implementada para Linux e NetBSD. A implementação de Lunatik consiste em três componentes básicos: o interpretador apropriadamente embutido no *kernel*, para a execução de scripts Lua; uma interface de programação de *kernel* (KPI), usada pelos desenvolvedores de *kernel* para tornar scriptáveis os seus subsistemas; e uma interface de usuário para a carga e execução de scripts no interpretador Lua embutido no *kernel*. A parte executada dentro do *kernel* foi desenvolvida como módulo carregável.

4.2.1 Lua Embutido no Kernel

O principal componente do Lunatik é o interpretador Lua embutido no kernel do sistema operacional. Embora algumas mudanças tenham sido necessárias para embutir Lua nos kernels Linux e NetBSD, todas essas mudanças foram feitas de forma não-intrusiva, envolvendo somente a modificação de algumas macros no arquivo de cabeçalho de configuração de Lua e a substituição de algumas funcionalidades da biblioteca C padrão, as quais não estão disponíveis nesses kernels.

Além de resolver as dependências da biblioteca C padrão, nós também precisamos tratar o uso de tipos de ponto flutuante. Nós substituímos o tipo padrão numérico de Lua, definido como `double`, para o tipo inteiro `int64_t`; essa mudança demandou somente a redefinição de seis macros no arquivo de cabeçalho de configuração.

Além do interpretador Lua, nós também embutimos a biblioteca auxiliar Lua e algumas outras bibliotecas padrão que não dependem de recursos de sistema operacional ou tipos de ponto flutuante (as bibliotecas *coroutine*, *table* e *string*). Para embutir as bibliotecas padrão, não foi necessário modificá-las, apenas precisamos adicionar suporte para algumas funções da biblioteca C padrão que não estavam presentes nesses kernels. Para embutir a biblioteca auxiliar, foi necessário modificá-la para remover todo o código dependente de sistema operacional.

4.2.2 Kernel Programming Interface (KPI)

Scripts são carregados assincronamente no kernel a partir do espaço de usuário através de uma chamada de sistema, como descreveremos na seção 4.2.3. O tratador dessa chamada de sistema e o subsistema que desejamos prover scripting executam em fluxos de controle concorrentes; portanto, é necessário sincronizar o acesso aos estados Lua dentro do kernel. Além disso, precisamos sincronizar também o acesso a estados Lua compartilhados entre diferentes fluxos de controle no mesmo ou em diferentes subsistemas.

Nós provemos a sincronização de estados Lua dentro do kernel encapsulando os estados Lua em mecanismos de exclusão mútua definidos pelos clientes da KPI. A KPI Lunatik implementa uma estrutura de dados especial, chamada estado Lunatik, a qual encapsula um estado Lua dentro de um mecanismo de exclusão mútua definido por duas funções, uma para bloquear e outra para liberar acesso a um estado Lua dentro do kernel. Em vez de criar diretamente estados Lua, os clientes da KPI devem criar estados Lunatik. A função de

criação de estados Lunatik da KPI recebe como argumentos um alocador de memória e funções que definem os mecanismos de exclusão mútua. Para facilitar a criação de estados Lunatik, a KPI inclui um função auxiliar que usa os alocadores de memória e mecanismos de exclusão mútua de propósito geral providos por NetBSD (`kmem` e `kmutex`) e Linux (`kmalloc` e `spinlock`).

Para sincronizar o uso de estados Lua dentro do kernel, a KPI Lunatik provê uma função que intermedeia as chamadas à API C de Lua, usando o mecanismo de exclusão mútua definido no estado Lunatik. Embora essa função seja suficiente para todas as interações com o estado Lua encapsulado no estado Lunatik, nós incluímos funções auxiliares na KPI para facilitar algumas tarefas comuns, como a carga de bibliotecas e a execução de código Lua.

Lunatik fornece um registro global de estados dentro do *kernel*. Esse registro armazena estados e funções de controle de acesso aos estados, e é gerenciado por funções da KPI. Para tornar um estado Lunatik acessível a partir do espaço de usuário e/ou compartilhável com outros subsistemas, o subsistema cliente da KPI precisa registrar esse estado. Para registrar um estado Lunatik é necessário passar o estado, um identificador único (uma string) e uma função de controle de acesso ao estado, a qual é chamada quando o estado é acessado a partir do espaço de usuário.

O apêndice A apresenta uma descrição mais detalhada da interface de programação provida pela KPI Lunatik.

4.2.3 Interface de Usuário

A interface de usuário de Lunatik permite que usuários carreguem e executem dinamicamente scripts dentro do kernel a partir do espaço de usuário. Essa interface é composta por duas partes, uma que executa no espaço de usuário e outra que executa dentro do kernel. O componente que executa no nível de usuário consiste eu uma ferramenta de linha de comando — o comando Lunatik — e um arquivo descritor de um pseudo-dispositivo. O componente que executa no kernel implementa o *driver* do pseudo-dispositivo.

O comando Lunatik é bastante semelhante ao interpretador Lua standalone; ele recebe scripts para serem executados e pode também prover um prompt de comando interativo para os usuários. Quando um usuário executa o comando Lunatik, o comando invoca uma chamada de ioctl no pseudo-dispositivo correspondente. Essa chamada de sistema, por sua vez, invoca a função tratadora de ioctl registrada pelo driver do pseudo-dispositivo. A função tratadora possui dois comandos: um lista os estados registrados e outro carrega e executa scripts no kernel. É importante observar que qualquer aplicação de

usuário pode invocar diretamente o pseudo-dispositivo usando a chamada de sistema de ioctl, sem precisar utilizar a comando Lunatik.

Antes de carregar e executar o código Lua, o tratador de ioctl invoca a função de controle de acesso registrada no estado Lunatik para verificar a autorização do usuário para acessar esse estado. Se a função de controle de acesso retornar positivamente, a KPI é utilizada para carregar e executar o script de usuário no estado Lua embutido no kernel. Caso contrário, um erro de acesso é retornado para o chamador do comando ioctl.

4.3 Experimentos

Nesta seção demonstraremos o uso de Lunatik através de dois experimentos. O primeiro experimento, apresentado na seção 4.3.1, implementa o suporte para que um usuário defina, em tempo de execução, sua própria política para o gerenciamento de freqüência e voltagem do processador. Primeiramente, apresentaremos o subsistema responsável pelo gerenciamento de freqüência de *CPU* no Linux, Cpufreq, e, então, discutiremos o *scripting* desse subsistema através do Lunatik.

O segundo experimento, apresentado na seção 4.3.2, implementa o suporte para que processos de usuário criem chamadas de sistema especializadas através de *scripting* de *kernel* no NetBSD. Primeiramente, apresentaremos a idéia geral de composição de chamadas de sistema (conceito utilizado para a especialização de chamadas) e, então, discutiremos os resultados obtidos.

Além dos experimentos que realizamos, Lunatik também foi utilizado pelo Computer Networks Research Group da Universidade de Basel para prover manipulação e prototipação de protocolos de rede, através do *scripting* do subsistema Netfilter do Linux (30).

4.3.1 Cpufreq

Atualmente, a eficiência energética do processador é uma característica importante para os mais diversos tipos de sistemas de computação, como dispositivos móveis, computadores embarcados, *desktops*, servidores e *clusters* (31).

O consumo de energia nos microprocessadores depende diretamente da voltagem do núcleo e da freqüência de operação; mais precisamente, a energia consumida é proporcional quadraticamente à voltagem e linearmente à freqüência. A voltagem do núcleo, por sua vez, também depende diretamente da freqüência de operação; desta forma, para o microprocessador operar em uma baixa voltagem, também é necessário reduzir a sua freqüência. Baseada

nesta relação entre freqüência de operação, voltagem do núcleo e consumo de energia, grande parte dos microprocessadores atuais possui mecanismos para poupar energia através da mudança de freqüência e de voltagem em tempo de execução. As técnicas de mudança de freqüência e de voltagem são conhecidas, respectivamente, como escalonamento dinâmico de freqüência e de voltagem e foram inicialmente aplicadas em dispositivos móveis para estender a duração de bateria sem comprometer o desempenho.

O Cpufreq é um subsistema do *kernel* Linux que, baseado nos mecanismos de escalonamento citados acima, permite melhorar a eficiência energética do sistema através da regulagem, em tempo de execução, da freqüência e voltagem do processador. Esse subsistema é composto por três tipos de elementos: reguladores de freqüência *(governors)*, *drivers* de dispositivos e um núcleo. A organização do Cpufreq pode ser observada na Figura 4.3.

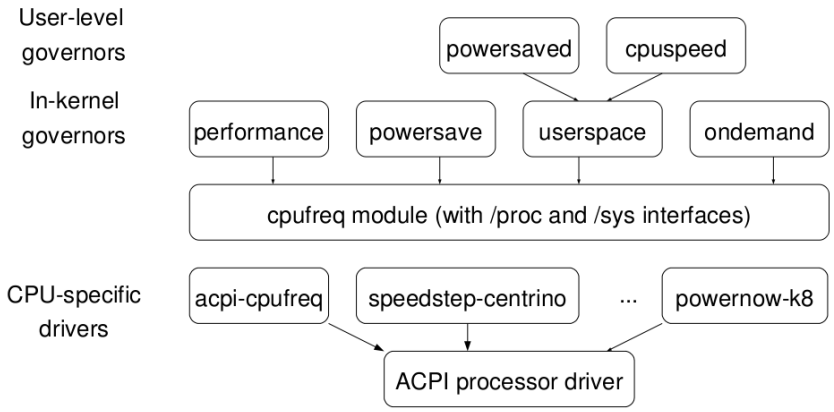

Figura 4.3: Organização do subsistema Cpufreq (31)

Os reguladores de freqüência são responsáveis pelas implementações de diferentes políticas de controle de freqüência do processador *(CPU)*. De forma geral, os reguladores de freqüência podem ser classificados em dois tipos: estáticos e dinâmicos. Reguladores estáticos definem apenas uma política geral de controle de freqüência, privilegiando, por exemplo, desempenho ou economia de energia. Os reguladores *performance* e *powersave*, que, respectivamente, mantém o processador na maior e menor freqüência disponíveis, são exemplos de implementações de reguladores estáticos embutidos no Cpufreq. Reguladores dinâmicos, por sua vez, baseiam sua política de controle de freqüência na avaliação periódica de algum indicador — tipicamente a utilização do processador — determinando, após cada avaliação, a freqüência adequada de operação. Um exemplo de regulador dinâmico embutido no Cpufreq é o *ondemand*, que, para balancear desempenho e eficiência energética, aumenta a freqüência de operação do processador quando sua taxa de utilização é alta,

diminuindo essa freqüência, para poupar energia, quando uma carga leve é observada (31).

Apenas um regulador de freqüência pode estar ativo no sistema em um dado momento. Um regulador é definido durante a inicialização do sistema, porém um outro regulador pode ser selecionado em tempo de execução.

Os *drivers* de dispositivos implementam os mecanismos de baixo nível necessários para o controle de freqüência e de voltagem da *CPU*. Esses mecanismos são específicos por plataforma; cada tecnologia diferente precisa de um *driver* específico. Como a voltagem do núcleo é dependente da freqüência de operação, o controle da voltagem é realizado pelo *driver* de acordo com a mudança de freqüência.

O núcleo do Cpufreq é responsável por intermediar a comunicação entre os reguladores e os *drivers*. Ambos, reguladores e *drivers*, devem registrar-se no núcleo e oferecer um conjunto pré-definido de interfaces. Através dessas interfaces, um regulador pode, por exemplo, solicitar ao *driver* que ele informe as freqüências de operação disponíveis ou que o microprocessador passe a operar em determinada freqüência. O núcleo do Cpufreq também implementa, via *sysfs*, uma interface que permite alterar, em tempo de execução, a seleção do regulador de freqüência ativo.

Além das políticas implementadas por seus reguladores embutidos, o Cpufreq pode incorporar dinamicamente outras opções de controle de freqüência, através da instalação de novos reguladores implementados como módulos do *kernel*. O Cpufreq permite também a incorporação de reguladores executados no espaço de usuário, disponibilizando, para isso, um regulador embutido especial (*userspace*) e um conjunto de interfaces via *sysfs*.

Os reguladores executados no espaço de usuário apresentam a vantagem de serem mais fáceis de serem implementados do que os reguladores embutidos no *kernel*; entretanto, eles exigem trocas de contexto a cada execução, o que acarreta em um maior dispêndio de recursos. Os reguladores embutidos no *kernel*, por sua vez, podem se beneficiar das próprias interfaces e estruturas de dados internas do *kernel*, além de não sofrer dos dispêndios de recursos vinculados a trocas de contexto.

Scripting do Cpufreq

Baseados na idéia de *scripting* de *kernel* e nas vantagens apresentadas para executarmos reguladores de freqüência dentro do *kernel*, construímos o suporte para "scriptar" o subsistema Cpufreq em Lua utilizando Lunatik. Nosso objetivo em oferecer este suporte é possibilitar que usuários definam as suas próprias regras para o controle de freqüência de *CPU* e as carreguem

dinamicamente para serem executadas dentro do *kernel*. Este suporte foi construído através da criação de um regulador de freqüência especial, chamado regulador Lunatik, que é responsável por chamar periodicamente uma função definida pelo script de usuário, o qual contém as regras reais de controle de freqüência.

Desenvolvemos também um script como demonstração do conceito apresentado, chamado Ondemand.lua. Este script é uma reimplementação simplificada do regulador Ondemand, que altera a freqüência de operação do processador, baseado na sua carga de utilização, com o objetivo de poupar energia minimizando a possível perda de desempenho ao diminuir a freqüência (31). A Figura 4.4 mostra em alto nível o algoritmo utilizado pelo Ondemand e, conseqüentemente, pelo Ondemand.lua. Esse algoritmo é executado periodicamente pelo regulador Ondemand para ajustar a freqüência dos processadores presentes no sistema.

```
para todo processador no sistema, faça:
  obtém a utilização desde a última verificação

  se (utilização > limite_superior), então:
    aumenta freqüência para o valor máximo

  senão se (utilização < limite_inferior), então:
    diminui freqüência em 20%
  fim "se"
fim "para todo"
```

Figura 4.4: Algoritmo do regulador Ondemand (31)

A Figura 4.5 mostra a implementação completa do regulador Ondemand.lua, contendo uma implementação para a função chamada periodicamente pelo regulador Lunatik para processar o controle da freqüência. Essa função é chamada através de um nome fixado no regulador Lunatik: `cpufreq.throttle`.

Para atingir o nosso objetivo de prover *scripting* para o subsistema Cpufreq, precisamos criar algumas bibliotecas para estender o interpretador Lua embutido com funcionalidades de subsistemas do *kernel*. Desta forma, criamos as seguintes bibliotecas de extensão: `cpufreq`, `cpumask` e `kernel_stat`.

```
1  up_threshold = 80
2  down_threshold = 30
3
4  function cpufreq.throttle()
5    -- para todo processador no sistema, faça:
6    for cpu = 1, cpumask.num_online_cpus() do
7      local policy = cpufreq.get_policy(cpu)
8
9      -- obtém a utilização desde a última verificação
10     local load = get_load(cpu)
11
12     if (load > up_threshold) then
13       -- aumenta a freqüência para o valor máximo
14       cpufreq.driver_target(cpu, policy.max, "high")
15
16     elseif(load < down_threshold) then
17       -- diminui a freqüência em 20%
18       local new_freq = policy.cur * 80 / 100
19       cpufreq.driver_target(cpu, new_freq, "low")
20     end
21   end
22 end
23
24 wall = 0; wall_prev = 0
25 idle = 0; idle_prev = 0
26
27 function get_load(cpu)
28   local cpustat = kernel_stat.get_cpustat(cpu)
29
30   for , ticks in ipairs(cpustat) do
31     wall = wall + ticks
32   end
33
34   idle = cpustat.idle + cpustat.iowait + cpustat.nice
35
36   local wall_delta = wall - wall_prev
37   local idle_delta = idle - idle_prev
38
39   local busy_delta = wall_delta - idle_delta
40
41   local load = (100 * busy_delta) / wall_delta
42
43   wall_prev = wall
44   idle_prev = idle
45
46   return load
47 end
```

Figura 4.5: Implementação do regulador Ondemand em Lua (Ondemand.lua)

A biblioteca `cpufreq` é utilizada para exportar as funções necessárias para o controle da freqüência da CPU. Ela disponibiliza as funções `get_policy` e `driver_target`. A função `get_policy` é utilizada para obter algumas informações referentes ao controle de freqüência de uma *CPU*, como a freqüência corrente e a máxima e a mínima freqüências suportadas. A função `driver_target` é utilizada para solicitar ao *driver* da *CPU* a mudança de sua freqüência. Para utilizarmos a função `driver_target`, é necessário fornecer o identificador da *CPU*, a freqüência-alvo e um parâmetro adicional. Este parâmetro adicional indica se desejamos utilizar a freqüência disponível imediatamente acima ou abaixo da freqüência-alvo solicitada.

A biblioteca `cpumask` disponibiliza apenas a função `num_online_cpus`, a qual retorna o número de *CPUs* disponíveis no sistema. Essa função é utilizada para percorrer todas as *CPUs* conforme descrito no algoritmo.

A biblioteca `kernel_stat` também disponibiliza apenas uma função, a `get_cpu_stat`, utilizada pela função auxiliar `get_load` para calcular a calcular a carga de utilização do processador durante o tempo decorrido entre as chamadas desta função. A função `get_cpu_stat` exporta para Lua uma estrutura de dados do escalonador de processos do Linux utilizada para fornecer estatísticas de utilização do processador, chamada `cpu_stat`. Esta estrutura de dados fornece os períodos de tempo em que o processador permaneceu ativo ou inativo desde a sua inicialização, separados por tipo de atividade ou inatividade. Mapeamos a estrutura de dados `cpu_stat` através de uma tabela Lua.

A implementação do script Ondemand.lua também permite que ela própria seja adaptada. Ela define duas variáveis globais (`up_threshold` e `down_threshold`) responsáveis pelos limites superior e inferior do algoritmo. Através dessas variáveis é possível ajustar o comportamento do script utilizando, por exemplo, o prompt de comando interativo de Lunatik.

Resultados

Com o objetivo de atestar a viabilidade de *scripting* do Cpufreq usando Lunatik, medimos o desempenho da execução do script Ondemand.lua. Para isso, medimos o tempo médio em 5.000 execuções da função `cpufreq.throttle` implementada nesse script. O tempo medido foi de 8 μs. Utilizamos um processador Intel Pentium M (1,6 GHz) para esse experimento.

O regulador Ondemand original utiliza o valor de 200 vezes a latência do processador modificar a freqüência de operação como o tempo de intervalo entre chamadas. Utilizamos o mesmo cálculo para determinar o intervalo entre chamadas para o regulador Lunatik (e, conseqüentemente, para o script

Ondemand.lua). No caso do processador utilizado nesse experimento, o valor utilizado como intervalo foi de 20.000 μs. Desta forma, o tempo de execução de `cpufreq.throttle` foi de 0,04% do intervalo entre chamadas. Isto significa que o processador passa um tempo igual a 0,04% do intervalo entre chamadas ocupado com a execução do script Ondemand.lua. Portanto, consideramos o *scripting* do Cpufreq usando Lunatik bastante viável em termos de desempenho.

Além da viabilidade quanto ao desempenho, também analisamos a facilidade de desenvolvimento do Ondemand.lua em comparação com a implementação original do regulador Ondemand em C. Estabelecemos como parâmetro a quantidade de linhas de código das duas implementações. A tabela 4.3.1 mostra em detalhes a quantidade de linhas de código da implementação original e da nossa implementação em Lua. Embora a comparação baseada exclusivamente em quantidade de linhas de código não seja perfeita, ela nos dá um indicador razoável da complexidade de desenvolvimento, sobretudo, pela discrepância entre as duas quantidades. O script Ondemand.lua tem menos de 7% das linhas de código do original em C. A implementação completa utilizando Lunatik, incluindo o regulador Lunatik, as bibliotecas de extensão necessárias e o script em Lua, é inferior a 37% do regulador Ondemand original em linhas de código. Desta forma, concluímos que o Ondemand.lua é bastante viável. Contudo, é importante destacar que o Ondemand.lua é uma implementação simplificada do Ondemand original e se atém somente a implementar o algoritmo básico de funcionamento. O Ondemand original possui ainda outras funções, como por exemplo, interagir com o Sysfs.

Descrição	Arquivo	Linhas de Código
Ondemand original	cpufreq_ondemand.c	713
Ondemand em Lua	ondemand.lua	47
Regulador Lunatik	cpufreq_lunatik.c	82
Bibliotecas de extensão	—	129
Total usando Lunatik	—	258

Tabela 4.1: Detalhamento das quantidades de linhas de código da implementação original do regulador Ondemand e do Ondemand.lua (usando Lunatik)

4.3.2 Composição de Chamadas de Sistema

Composição de chamadas (CompositeCalls) (32, 33) é uma técnica utilizada com o objetivo de se evitar atravessamentos entre diferentes domínios de proteção. A idéia central desta técnica é compor várias chamadas em uma única, para, assim, evitar-se os atravessamentos (32). Essa técnica é também

chamada de lote de chamadas (*batching*) em outros trabalhos (34). A composição de chamadas é aplicada em diferentes tipos de domínios de proteção, como por exemplo chamadas a um servidor remoto através da rede. No nosso caso, estamos interessados na utilização dessa técnica para evitarmos o atravessamento causado por chamadas ao sistema operacional, ou seja, trocas de contexto do processador. Chamamos, então, esse caso de *Composição de Chamadas de Sistema*.

Como exemplo, considere um programa que copia dados de um arquivo para outro. A figura 4.6 mostra uma função, chamada `cp`, que implementa a cópia de dados entre dois arquivos em Lua. Essa função realiza uma chamada de sistema a cada chamada às funções `read` ou `write`. Conseqüentemente, a cada chamada de sistema ocorre um atravessamento entre domínios que, por sua vez, freqüentemente envolve duas trocas de contexto de *CPU* (uma para entrar no modo *kernel* e outra para retornar ao espaço de usuário). Além das trocas de contexto de *CPU*, em algumas plataformas, é necessário copiar os dados entre os dois domínios de proteção (no caso, espaços de endereçamento).

```
1 function cp(fin, fout)
2   while true do
3     local buffer = fin:read("*line")
4     if not buffer then break end
5     fout:write(buffer)
6   end
7 end
```

Figura 4.6: Script Lua para a cópia de dados entre dois arquivos no espaço de usuário (usando chamadas de sistema)

Para se utilizar essa técnica, compomos várias chamadas de sistema em uma única. Assim, no exemplo da função `cp`, mostrada na figura 4.6, seria necessário estender o *kernel* do sistema operacional com a função `cp` (ou uma similar com o mesmo propósito), adicionando uma nova chamada de sistema ao SO. Desta forma, realiza-se apenas um atravessamento de domínio, no lugar dos vários realizados originalmente.

Scripting de Chamadas de Sistema

Veremos como utilizar *scripting* de *kernel* para a realização da composição de chamadas de sistema. A idéia geral é exportar para os scripts as funções presentes na tabela de chamadas de sistema do SO, sob a forma de biblioteca de ligação. Assim, os scripts podem ser utilizados para compor chamadas de sistemas. O *scripting*, nesse caso, é feito através do caso de extensão

do interpretador. Chamamos essa forma de composição de chamadas de sistema de *Scripting de Chamadas de Sistema*.

Baseamos essa abordagem de *Scripting de Chamadas de Sistema* no trabalho de Ballesteros et al. (32), que utiliza um interpretador de uma linguagem reduzida embutido em *kernel* para processar chamadas de sistema compostas. Esse interpretador é chamado de `interp` e foi implementado para dois sistemas operacionais, Linux e Off++ (35). A linguagem de `interp` possui o único propósito de permitir que usuários construam chamadas compostas através do seqüenciamento de comandos. Para isso, ela possui na sua interface de programação comandos equivalentes a chamadas de sistema e um comando de repetição (*while*). A principal diferença da nossa abordagem em relação a de Ballesteros et al. é o uso de uma linguagem de script, com todas as vantagens previamente debatidas, no lugar de uma linguagem reduzida, rudimentar e de caráter específico. Por exemplo, a linguagem utilizada em `interp` não possui controles de fluxo, como *if-then-else*, funções ou estruturas de dados complexas.

A seguir listamos alguns exemplos de aplicações de composição de chamadas (33) que podem ser tratadas por *scripting* de chamadas de sistema:

Melhoria de latência. Este é o principal uso de composição de chamadas: melhorar a latência através da redução de atravessamentos entre domínios de proteção. As chamadas de sistemas normalmente providas por sistemas operacionais são bastante básicas (e.g., `read`, `write`, `open`). Aplicativos costumam executar diversas chamadas de sistema para realizar tarefas específicas (e.g., função `cp` da figura 4.6). *Scripting* de chamadas de sistema possibilita a implementação de novas chamadas de sistema especializadas com o objetivo de se reduzir a latência em tarefas específicas.

E/S Vetorizada. Entrada e saída vetorizada é uma técnica utilizada em alguns sistemas operacionais para possibilitar leitura ou escrita utilizando múltiplos *buffers*. Para isso, o SO deve fornecer chamadas de sistemas especiais como `readv` e `writev`, presentes na especificação POSIX. *Scripting* de chamadas de sistema possibilita a implementação de tais chamadas.

Chamadas adiadas. Esta aplicação consiste em carregar o script que fará as chamadas de sistema e agendá-lo para execução sem aguardar o resultado. O resultado da execução poderia ser obtido posteriormente utilizando recursos de programação como *futuros* ou *promessas*.

Experimento de Scripting de Chamadas de Sistema

Como prova do conceito de *scripting* de chamadas de sistema, elaboramos um experimento utilizando Lunatik no sistema NetBSD. Esse experimento consistiu no uso de um script Lua para a implementação de um padrão de programação utilizado em programas que fazem cópias de dados entre dois arquivos, como por exemplo os comandos `cat`, `cp`, `tar` e `dd` presentes em ambientes tipo-UNIX. Esse padrão foi ilustrado pela função `cp` apresentada na figura 4.6. Nesse padrão, são feitas chamadas de sistema e, conseqüentemente, trocas de contexto a cada chamada de função `read` ou `write`. A idéia desse experimento é carregar e executar no *kernel* um script semelhante ao apresentado na figura 4.6, contendo um laço com chamadas as implementações internas das chamadas de sistema `read` e `write`. Desta forma, torna-se necessário apenas uma chamada de sistema para carregar o script contendo a função com o laço e outra para executá-la e, conseqüentemente, efetuar a cópia entre os arquivos. Assim, evita-se as várias trocas de contexto inerentes a implementação original desse padrão de programação.

Para possibilitar esse experimento, precisamos primeiro desenvolver uma biblioteca de ligação para as implementações internas das chamadas de sistema a serem efetuadas pelo script (no caso, as funções `sys_read` e `sys_write`). Essas funções, por sua vez, utilizam uma outra função para efetuar a leitura ou a escrita em cada caso: `dofileread` e `dofilewrite`. Para o desenvolvimento da biblioteca de ligação, precisamos modificar ligeiramente a implementação das funções `dofileread` e `dofilewrite` para possibilitar a utilização de um *buffer* alocado no próprio *kernel*, no lugar de um alocado no espaço de usuário. Utilizamos essas modificações para criar novas funções (`sys_read_` e `sys_write_`), em vez de substituir as implementações existentes dessas chamadas de sistema.

Nomeamos de `syscall` a biblioteca de ligação que criamos para esse experimento, pois ela exporta funções que estão na interface de chamadas de sistema do SO. Exportamos duas funções através dela: `syscall.read` e `syscall.write`. A função `syscall.read` faz a ligação com a função `sys_read_`; recebe como parâmetros um inteiro descritor de arquivo e o número de *bytes* a serem lidos; e retorna o número de *bytes* que foram efetivamente lidos. A função `syscall.write` faz a ligação com a função `sys_write_`; recebe como parâmetros um inteiro descritor de arquivo e o número de *bytes* a serem escritos; e retorna o número de *bytes* que foram efetivamente escritos. As funções `syscall.read` e `syscall.write` compartilham um *buffer* alocado internamente pela biblioteca `syscall`. Utilizamos para esse *buffer* o tamanho dado pela macro `MAXBSIZE`, a qual define o tamanho máximo em *bytes* dos

blocos de sistemas de arquivos em sistemas BSD.

Implementamos um script similar ao apresentado na figura 4.6 utilizando a biblioteca `syscall`. Mostramos esse script na figura 4.7. A função `cp`, nesse caso, recebe como parâmetros os descritores de arquivo de entrada e saída (`fin` e `fout`, respectivamente) e o número de bytes a serem copiados (`to_copy`). Utilizamos esse terceiro parâmetro para facilitar nossos experimentos, no caso em que não desejamos copiar inteiramente o conteúdo de um arquivo. Desta forma, quando um processo de usuário carrega a função `cp` no *kernel*, ele estende o SO criando uma chamada de sistemas especializada para a cópia de arquivos.

```
1  function cp(fin, fout, to_copy)
2    local count = 0
3    while count < to_copy do
4      local num_read = syscall.read(fin, MAXBSIZE)
5      if num_read == 0 then break end
6
7      syscall.write(fout, number_read)
8      count = count + num_read
9    end
10 end
```

Figura 4.7: Script Lua para a cópia de dados entre arquivos no *kernel* (*scripting* de chamadas de sistema)

Resultados

Com o objetivo de atestar o ganho de desempenho relacionado ao uso de *scripting* de chamadas de sistema com Lunatik, comparamos o desempenho da utilização do script apresentado na figura 4.7 com a utilização do método tradicional que realiza várias chamadas de sistema para a cópia de dados entre arquivos. Para isso, escrevemos dois programas em C: um utilizando as chamadas de sistemas convencionais `read` e `write` e outro utilizando Lunatik para realizar o *scripting* de chamadas de sistema com o script mostrado na 4.7.

Medimos os tempos de execução de ambos os métodos para diferentes quantidades de dados copiados. Os tempos foram medidos com base na média de 10.000 execuções. Utilizamos o programa `/usr/bin/time` do NetBSD para a medição dos tempos de execução. Esse programa separa os tempos medidos em duas porções: sistema, quantidade de tempo executada pelo processo em *kernel*; e usuário, quantidade de tempo executada pelo processo em espaço de usuário. A tabela 4.3.2 mostra os tempos de execução para cada método e os ganhos de *scripting* de chamadas de sistema em relação ao uso de chamadas de sistema convencionais para a mesma quantidade de dados. Realizamos essas

medidas em um Intel Celeron 743 (1,3 GHz) com 2 GB de memória e utilizamos um sistema de arquivos em memória (Tmpfs) para minimizar os efeitos dos tempos de acesso ao disco rígido nessas medidas.

Método	Tamanho		Tempo médio (ms)		Ganho (%)	
	Bytes	MAXBSIZE	Usuário	Sistema	Usuário	Sistema
chamadas de sistema	64 KB	1	0,62	0,79	—	—
scripting de chamadas			0,63	0,80	1,59	-1,25
chamadas de sistema	128 KB	2	0,68	0,87	—	—
scripting de chamadas			0,66	0,88	2,94	-1,14
chamadas de sistema	256 KB	4	0,80	1,08	—	—
scripting de chamadas			0,81	1,10	-1,24	-1,82
chamadas de sistema	512 KB	8	1,03	1,58	—	—
scripting de chamadas			1,00	1,44	2,91	8,86
chamadas de sistema	1 MB	16	1,25	2,32	—	—
scripting de chamadas			1,25	2,20	0	5,17
chamadas de sistema	2 MB	32	1,46	4,32	—	—
scripting de chamadas			1,46	4,04	0	6,48
chamadas de sistema	4 MB	64	0,30	12,8	—	—
scripting de chamadas			0,22	12,3	26,7	3,91
chamadas de sistema	8 MB	128	0,40	24,4	—	—
scripting de chamadas			0,28	23,6	30,0	3,28
chamadas de sistema	16 MB	256	0,54	47,7	—	—
scripting de chamadas			0,33	46,2	38,9	3,15
chamadas de sistema	32 MB	512	0,71	94,6	—	—
scripting de chamadas			0,33	91,6	53,5	3,17
chamadas de sistema	64 MB	1024	1,14	189,0	—	—
scripting de chamadas			0,34	183,1	70,1	3,12
chamadas de sistema	128 MB	2048	1,83	311,3	—	—
scripting de chamadas			0,31	303,1	83,1	2,63

Tabela 4.2: Resultados da execução dos métodos de *scripting* de chamadas de sistema e de chamadas de sistema convencionais

Utilizamos como tamanho para os *buffers* de ambos os métodos o valor de `MAXBSIZE`, que é igual a 64 KB para a plataforma utilizada nesse experimento. O programa que utiliza o método de chamadas de sistema convencionais realiza uma chamada `read` e uma `write` a cada `MAXBSIZE` *bytes* copiados. Desta forma, o número de `MAXBSIZE` unidades copiadas nos dá o número de chamadas de sistema efetuadas para `read` e `write` para o método convencional.

Os resultados indicam o ganho do tempo de `sistema` no método de *scripting* de chamadas a partir de 8 `MAXBSIZE`. Isso se dá, provavelmente, porque o tempo gasto com as trocas de contexto para um número inferior a 16 chamadas de sistema (6 para `read` e 6 para `write`) deve ser inferior a sobrecarga de se utilizar o interpretador Lua para executar o script de chamadas.

Observamos também um ganho bastante significativo (de 26,7% a 83,1%) no tempo de `usuário` no *scripting* de chamadas a partir de 64 `MAXBSIZE`. Isso se deve, provavelmente, pelo aumento do tempo total da cópia, devido a quantidade de *bytes* copiados. Ao aumentar-se o tempo de cópia, o processo

é escalonado mais vezes pelo SO. No caso do método convencional, esse efeito é acentuado, pois a cada troca de contexto devido as chamadas de sistema o processo pode ser preemptado para dar lugar a outro. No caso de *scripting* de chamadas, o processo é interrompido menos freqüentemente. Outro fator que colabora para o ganho de tempo de `usuário`, no caso de *scripting* de chamadas, é o fato do laço principal do programa (cópia de dados) ser executado dentro do *kernel*.

Outro efeito que observamos foi o ganho modesto do tempo de `sistema` em relação ao tempo de `usuário` a partir de 64 `MAXBSIZE`. Isso também se deve, possivelmente, pelo tempo prolongado de execução total. Quando um processo é executado por um período consideravelmente longo de tempo, ele é escalonado várias vezes devido ao término da sua fatia de tempo. Assim, esse processo sofre várias trocas de contexto. Outro fator que possivelmente colabora com esse efeito é o tempo gasto pelo sistema de arquivos para copiar efetivamente os dados entre os *buffers* internos do sistema operacional e entre o *buffer* que utilizamos para a cópia. Esse tempo, gasto pelo sistema de arquivos para a manipulação dos dados. aparenta dominar o tempo de `sistema`.

Para verificarmos o efeito causado pela manipulação de dados feita pelo sistema de arquivos, realizamos medidas complementares ao nosso teste inicial. Essas medidas complementares foram feitas utilizando pseudo-dispositivos, que são arquivos especiais que não são tratados por um sistema de arquivos. Assim, realizamos a cópia de dados entre os pseudo-dispositivos `/dev/zero`, o qual apenas retorna a quantidade de zeros desejada, e `/dev/null`, o qual apenas descarta os *bytes* de entrada. A tabela 4.3.2 mostra os nossos resultados para esse caso. Executamos esse experimento no mesmo ambiente do anterior e também baseamos as medidas de tempo na média de 10.000 execuções.

No caso dos pseudo-dispositivos, notamos um comportamento mais homogêneo e mais diretamente relacionado as trocas de contexto decorrente das chamadas de sistema por não ter a interferência do sistema de arquivos.

Baseados nos resultados desse experimento, concluímos que é viável o uso de *scripting* de chamadas de sistema usando Lunatik.

Método	Tamanho		Tempo médio (ms)		Ganho (%)	
	Bytes	MAXBSIZE	Usuário	Sistema	Usuário	Sistema
chamadas de sistema	64 KB	1	0,55	0,70	—	—
scripting de chamadas			0,55	0,70	0	0
chamadas de sistema	128 KB	2	0,56	0,71	—	—
scripting de chamadas			0,57	0,72	-1,72	-1,40
chamadas de sistema	256 KB	4	0,58	0,74	—	—
scripting de chamadas			0,58	0,74	0	0
chamadas de sistema	512 KB	8	0,62	0,79	—	—
scripting de chamadas			0,61	0,76	1,61	3,80
chamadas de sistema	1 MB	16	0,71	0,95	—	—
scripting de chamadas			0,65	0,83	8,50	1,26
chamadas de sistema	2 MB	32	0,84	1,15	—	—
scripting de chamadas			0,71	0,96	15,5	16,5
chamadas de sistema	4 MB	64	1,03	1,57	—	—
scripting de chamadas			0,86	1,22	16,5	22,3
chamadas de sistema	8 MB	128	1,32	2,62	—	—
scripting de chamadas			1,14	1,77	13,6	32,4
chamadas de sistema	16 MB	256	1,42	5,20	—	—
scripting de chamadas			1,37	3,10	3,52	40,4
chamadas de sistema	32 MB	512	0,45	11,5	—	—
scripting de chamadas			1,16	6,46	-61,2	43,8
chamadas de sistema	64 MB	1024	0,75	22,1	—	—
scripting de chamadas			0,23	13,6	69,3	38,5
chamadas de sistema	128 MB	2048	1,35	43,2	—	—
scripting de chamadas			0,28	26,2	79,3	39,4

Tabela 4.3: Resultados da execução dos métodos de *scripting* de chamadas de sistema e de chamadas de sistema convencionais usando pseudo-dispositivos

5
Conclusão

Nesta dissertação, apresentamos os conceitos de sistema operacional scriptável e de *scripting* de *kernel*, os quais são resultados da aplicação das idéias de extensibilidade através de *scripting* e de desenvolvimento de programas utilizando linguagens de script aos conceitos de sistema operacional e sistema operacional extensível.

Nós implementamos Lunatik, a nossa infra-estrutura que provê suporte para scripting de kernel de sistema operacional com Lua, permitindo o desenvolvimento e extensão de *kernels* usando scripts Lua. Usando Lunatik, desenvolvedores de kernel podem tornar subsistemas "scriptáveis" e usuários podem carregar e executar scripts dinamicamente em um interpretador embutido no *kernel*. Lunatik é uma prova de conceito de que é possível prover um ambiente para o *scripting* de *kernel* similar àqueles para aplicações em espaço de usuário.

Nós acreditamos que nossa abordagem para *scripting* de *kernel* pode auxiliar na mitigação dos problemas conhecidos de sistemas operacionais: flexibilidade, confiabilidade, desempenho e facilidade de desenvolvimento.

Acreditamos também que essa abordagem para *scripting* de *kernel* pode auxiliar na inovação em desenvolvimento de *kernel* de sistema operacional. Primeiro, ela disponibiliza um ambiente de programação interpretado e de mais alto nível para o *kernel*. Portanto, ela torna a prototipação e experimentação mais rápidas. Segundo, ela disponibiliza uma plataforma de extensibilidade mais fácil e ágil do que módulos de kernel e mais flexível do que ajuste de parâmetros. Portanto, em subsistemas previamente adaptados para serem scriptados, ela permite que não-desenvolvedores de *kernel*, como desenvolvedores de aplicações e administradores de sistemas, possam fazer experimentos com o *kernel* para atender as suas próprias necessidades.

A implementação de Lunatik para NetBSD foi desenvolvida, em parte, dentro do programa *Google Summer of Code*. Essa implementação está acessível em `http://netbsd-soc.sourceforge.net/projects/luakern/`, junto com um relatório sobre o desenvolvimento, um pequeno tutorial e um manual de uso da interface de programação de *kernel*. Atualmente, Lunatik está

sendo desenvolvido no escopo do projeto NetBSD para possivelmente ser incorporado nesse sistema operacional, no futuro[1]. A implementação de Lunatik para Linux está acessível em `http://sourceforge.net/projects/lunatik/`.

A seguir discutiremos algumas lições sobre *scripting* de *kernel* que aprendemos durante o desenvolvimento desta dissertação.

5.1 Lições Aprendidas

Ao longo do desenvolvimento das implementações de Lunatik para Linux e NetBSD, pudemos aprender algumas lições sobre a implantação de *scripting* de *kernel*:

- A principal lição que tivemos é que adaptar um *kernel* para ser scriptado é muito mais complexo do que simplesmente embutir o interpretador de uma linguagem de script nele. Isso decorre, sobretudo, do fato de que a tarefa de desenvolver e estender *kernels* em si é bastante complexa.
- Existem diferentes contextos de execução em um *kernel* e isso impacta diretamente no desenvolvimento do interpretador da linguagem de script. Por exemplo, são necessários diferentes alocadores de memória, dependendo do contexto de execução.
- A cópia de grandes quantidades memória pode ser bastante custosa. Por isso, é importante aproveitar as implementações internas de estruturas de dados do *kernel* sempre que possível. Por exemplo, ao implementarmos o exemplo de filtragem de dados em *kernel* do capítulo 4 (função `filter` da figura 4.1), obtivemos uma grande perda de desempenho ao copiarmos dados criando novas *strings* em Lua (usando a função `lua_pushstring` da API de Lua). Nesse caso, o mais indicado seria mapear os dados (e.g., em um *user datum*) para serem acessados em Lua.
- A criação de uma infra-estrutura de *scripting* de *kernel* também é mais complexa do que simplesmente embutir um interpretador no *kernel*. São necessários mecanismos para a carga de scripts e formas de integrar a linguagem de script com o *kernel*.
- O *scripting* de *kernel* se posiciona entre o ajuste de parâmetros em tempo de execução (e.g., Sysctl e Sysfs) e a criação de módulos de *kernel*. É importante buscar aplicações que caibam nesse nicho, utilizando a ferramente correta (ajuste de parâmetros, módulos de *kernel* ou *scripting* de *kernel*) para cada necessidade.

[1]Anúncio pode ser visualizado em `http://netbsd.org/changes/index.html#newdev201101`

Referências Bibliográficas

[1] LAMPSON, B. W. On reliable and extendible operating systems. In: *Proceedings of the Second NATO Conference on Techniques in Software Engineering*. [S.l.: s.n.], 1969. 1, 2.1.1, 2.2, 2.2.1

[2] ENGLER, D. R.; KAASHOEK, M. F. Exterminate all operating system abstractions. In: USENIX. *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*. [S.l.], 1995. p. 78–83. ISBN 0818670819. 1, 2.2.1

[3] BERSHAD, B. N. et al. SPIN — an extensible microkernel for application-specific operating system services. *ACM SIGOPS Operating Systems Review*, ACM, v. 29, n. 1, p. 74–77, 1995. ISSN 0163-5980. 1, 2.1.1, 2.2, 2.2.1

[4] SMALL, C.; SELTZER, M. *VINO: An Integrated Platform for Operating System and Database Research*. [S.l.], 1994. 1, 2.2, 2.2.1, 2.2.1

[5] HUNT, G. C.; LARUS, J. R. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, ACM, v. 41, n. 2, p. 37–49, 2007. ISSN 0163-5980. 1, 2.1.1, 2.2

[6] SAVAGE, S.; BERSHAD, B. N. Issues in the design of an extensible operating system. In: *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*. [S.l.: s.n.], 1994. 1, 2.2, 2.2.1, 2.2.1, 2.2.1

[7] OUSTERHOUT, J. K. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, v. 31, n. 3, p. 23–30, 1998. 1, 2.3

[8] IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; FILHO, W. C. Lua — an extensible extension language. *Software: Practice and Experience*, John Wiley & Sons, v. 26, n. 6, p. 635–652, 1996. 1, 2.3.1

[9] VIEIRA NETO, L. *Repensando o Conceito de Sistema Operacional*. dez. 2008. Projeto Final de Graduação em Engenharia de Computação, Departamento de Informática, PUC-Rio. 1

[10] HANSEN, P. B. *Operating System Principles*. [S.l.]: Prentice Hall, 1973. 2.1

[11] TANENBAUM, A. S.; WOODHULL, A. S. *Operating Systems Design and Implementation*. Third. [S.l.]: Prentice Hall, 2006. 2.1

[12] STALLINGS, W. *Operating Systems: Internals and Design Principles*. [S.l.]: Prentice Hall, 2008. 2.1

[13] ENGLER, D. R.; KAASHOEK, M. F.; O'TOOLE, J. W. Exokernel: An operating system architecture for application-level resource management. In: ACM. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. [S.l.], 1995. p. 251–266. ISBN 0897917154. 2.1.1, 2.2, 2.2.1, 3.3

[14] DIJKSTRA, E. W. The structure of the "THE"-multiprogramming system. In: ACM. *Proceedings of the first ACM symposium on Operating System Principles*. [S.l.], 1967. p. 10–1. 2.2, 2.2.1

[15] ENGLER, D. R. *The Exokernel Operating System Architecture*. Tese (Doutorado) — MIT, 1998. 2.2

[16] SELTZER, M. I. et al. Issues in extensible operating systems. *Computer Science Technical Report TR-18-97*, Harvard University, 1997. 2.2, 2.2.1

[17] CAMPBELL, R. H.; TAN, S. M. μchoices: an object-oriented multimedia operating system. In: USENIX. *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*. [S.l.], 1995. p. 90–94. 2.2, 2.2.1, 3.3

[18] MAHESHWARI, U. Extensible operating systems. *Area Exam Report*, MIT, 1994. 2.2.1

[19] MONTZ, A. B. et al. Scout: A communications-oriented operating system. In: USENIX. *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*. [S.l.], 1995. p. 58–61. 2.2.1

[20] MOCHEL, P.; MURPHY, M. *Sysfs — The Filesystem for Exporting Kernel Objects*. 2009. `http://kernel.org/doc/Documentation/filesystems/sysfs.txt`. 2.2.1

[21] THE NETBSD FOUNDATION. *NetBSD Kernel Developer's Manual*. [S.l.]. 2.2.1

[22] ESPOSITO, D. *Windows 2000 Registry: Latest Features and APIs Provide the Power to Customize and Extend Your Apps*. 2000. `http://msdn.microsoft.com/en-us/magazine/bb985037.aspx`. 2.2.1

[23] BARHAM, P. et al. Xen and the art of virtualization. In: ACM. *Proceedings of the nineteenth ACM symposium on Operating systems principles*. [S.l.], 2003. p. 164–177. 2.2.1

[24] LEFKOWITZ, G. *Extending Vs. Embedding — There Is Only One Correct Decision*. 2003. `http://twistedmatrix.com/users/glyph/rant/extendit.html`. 2.3.1

[25] MUHAMMAD, H.; IERUSALIMSCHY, R. C APIs in extension and extensible languages. *Journal of Universal Computer Science*, v. 13, n. 6, p. 839–853, 2007. 2.3.1

[26] IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; CELES, W. The evolution of Lua. In: ACM. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. [S.l.], 2007. p. 2–26. 2.3.1

[27] IERUSALIMSCHY, R. *Programming in Lua*. [S.l.]: Lua.org, 2006. 2.3.1

[28] INTERNATIONAL STANDARDIZATION ORGANIZATION. *ISO/IEC 9899:1999(E); Programming languages — C*. Second. [S.l.], 1999. 2.3.1

[29] CONWAY, C. L.; EDWARDS, S. A. NDL: a domain-specific language for device drivers. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2004. v. 39, n. 7, p. 30–36. 3.2

[30] GRAF, A. *PacketScript — a Lua Scripting Engine for in-Kernel Packet Processing*. Dissertação (Mestrado) — Computer Science Department, University of Basel, jul. 2010. 4.3

[31] PALLIPADI, V.; STARIKOVSKIY, A. The ondemand governor. In: *Proceedings of the Linux Symposium*. [S.l.: s.n.], 2006. v. 2, p. 215–230. (document), 4.3.1, 4.3, 4.3.1, 4.3.1, 4.4

[32] BALLESTEROS, F. J. et al. Using interpreted compositecalls to improve operating system services. *Software Practice and Experience*, v. 30, n. 6, p. 589–615, 2000. 4.3.2, 4.3.2

[33] PATINO, M. et al. Compositecalls: A design pattern for efficient and flexible client-server interaction. In: *Proceedings of PloP*. [S.l.: s.n.], 1999. 4.3.2, 4.3.2

[34] BALLESTEROS, F. et al. Batching: A design pattern for efficient and flexible client/server interaction. *Transactions on Pattern Languages of Programming I*, Springer, p. 48–66, 2009. 4.3.2

[35] BALLESTEROS, F. et al. Object orientation in off++ a distributed adaptable μkernel. In: *Proceedings of the ECOOP Workshop on Object Orientation and Operating Systems*. [S.l.: s.n.], 1999. p. 49–53. 4.3.2

A
Lunatik Kernel Programming Interface (KPI)

Neste apêndice apresentaremos a interface de programação de *kernel* que desenvolvemos para Lunatik, a qual chamamos de Lunatik *KPI (Kernel Programming Interface)*. A Lunatik KPI fornece suporte para auxiliar desenvolvedores a adaptarem partes do *kernel* para serem scriptadas usando Lua.

Apresentaremos, nesse apêndice, apenas a KPI desenvolvida para NetBSD. A implementação para Linux, por ter sido a primeira a ser desenvolvida, é uma versão simplificada da KPI para NetBSD.

A KPI para NetBSD será apresentadas sob a forma de páginas de manual UNIX, contendo as seguintes subseções:

Nome: nome da interface de programação que será abordada na página de manual, seguido de uma breve descrição sobre ela.

Sinopse: lista dos protótipos das funções disponíveis na interface e dos arquivos de cabeçalho que são necessários para a utilização delas.

Descrição: descrição do funcionamento das funções presentes na interface.

A.1
Lunatik KPI para NetBSD

Nesta seção, apresentaremos o manual da Lunatik KPI para NetBSD.

NOME

Lunatik KPI — interface de programação de kernel para Lunatik.

SINOPSE

```
#include <lua/lua.h>
#include <sys/lunatik.h>

lunatik_State *
lunatik_newstate(lua_Alloc allocf, void *ud, void *mutex,
```

```
    lunatik_lock_t lockf, lunatik_unlock_t unlockf);

void
lunatik_close(lunatik_State *Lk);

inline void
lunatik_lock(lunatik_State *Lk);

inline void
lunatik_unlock(lunatik_State *Lk);

inline void *
lunatik_alloc(lunatik_State *Lk, size_t size);

inline void
lunatik_free(lunatik_State *Lk, void *ptr, size_t size);

inline lua_State *
lunatik_getluastate(lunatik_State *Lk);

int
lunatik_runstring(lunatik_State *Lk, const char *str,
    char **resultp, size_t *size_resultp);

int
lunatik_runcfunction(lunatik_State *Lk, lua_CFunction cfunction,
    void *lightuserdata, char **resultp, size_t *size_resultp);

int
lunatik_registerstate(lunatik_State *Lk, const char *name);

#include <sys/kmem.h>
#include <sys/mutex.h>

lunatik_State *lunatikL_newstate(km_flag_t km_flags,
    kmutex_t *kmutex, kmutex_type_t kmutex_type);
```

DESCRIÇÃO

Lunatik é uma infra-estrutura para scripting de kernel usando Lua. Este manual descreve a interface de programação de kernel dessa infra-estrutura, a Lunatik KPI. Esta interface é um dos componentes da infra-estrutura Lunatik, a qual fornece suporte para que partes do kernel sejam adaptadas para serem scriptadas.

lunatik_newstate() cria um novo estado Lunatik. Um estado Lunatik é um estado Lua encapsulado por um mecanismo de sincronização. Em caso de sucesso essa função retorna o novo estado criado. Caso não tenha memória suficiente disponível no sistema, ela retorna `NULL`. Essa função toma os seguintes argumentos:

allocf ponteiro para função de alocação de memória. Esta função é utilizada para toda alocação de memória feita pelo Lunatik para esse estado e, conseqüentemente, por toda alocação de memória feita por Lua para o estado Lua encapsulado no estado Lunatik.

allocud ponteiro opaco passado para a função de alocação de memória, referenciada por `allocf`, a cada chamada.

mutex ponteiro opaco passado para as funções referenciadas por `lockf` e `unlockf`. Esse ponteiro é tipicamente usado como descritor de exclusão mútua para o mecanismo de sincronização. Desta forma, as funções referenciadas por `lockf` e `unlockf` são chamadas sobre um `mutex` para, respectivamente, bloqueá-lo ou desbloqueá-lo.

lockf ponteiro para a função de bloqueio do mecanismo de sincronização. Esta função toma como argumento o ponteiro `mutex`. Ela deve bloquear caso tenha sido chamada anteriormente sem que a função referenciada pelo argumento `unlockf` tenha sido chamada em seguida sobre o mesmo parâmetro `mutex`. Caso contrário, ela deve simplesmente retornar. Essa função deve obedecer a seguinte definição de tipo: `typedef void (*lunatik_lock_t)(void *)`.

unlockf ponteiro para função de desbloqueio do mecanismo de sincronização. Esta função toma como argumento o ponteiro `mutex`. Ela deve liberar as chamadas bloqueadas à função referenciada pelo argumento `lockf`. Caso existam uma ou mais chamadas pendentes à função de bloqueio sobre um dado `mutex` (ou seja, bloqueadas), uma (e somente uma) das chamadas pendentes deve ser li-

berada. Essa função deve obedecer a seguinte definição de tipo: `typedef void (*lunatik_unlock_t)(void *)`.

lunatik_close() encerra um estado Lunatik. Toda a memória alocada para esse estado, incluindo a memória utilizada diretamente por Lua, é liberada utilizando a função apontada por `allocf` (passado como parâmetro na criação do estado). Essa função toma o seguinte argumento:

Lk descritor do estado Lunatik que se deseja encerrar.

lunatik_runstring() executa um script Lua contido em uma string. Essa função toma os seguintes argumentos:

Lk descritor do estado Lunatik onde deseja-se executar o script Lua.

str string terminada em `'\0'` contendo o script Lua a ser executado.

resultp ponteiro para string passado como referência para receber o resultado retornado pelo script Lua. O valor de retorno será convertido para uma string. Caso não se deseje receber o resultado da execução, deve-se passar o valor `NULL` através deste parâmetro. Em caso de erro de compilação ou de execução do script, a string retornada conterá a mensagem de erro. A string contendo o resultado é alocada dinamicamente utilizando o alocador do estado `Lk`. É responsabilidade do cliente da interface desalocar essa string. Para facilitar esse processo, pode-se usar a função `lunatik_free`.

size_resultp ponteiro passado como referência para receber o tamanho da string de resultado.

lunatik_runcfunction() executa uma função C em um estado Lua passando como parâmetro um `light userdata`. Essa função equivale as chamadas das funções `lua_pushcfunction`, `lua_pushlightuserdata` e `lua_pcall` em seqüência. Essa função toma os seguintes argumentos:

Lk descritor do estado Lunatik onde deseja-se executar a função C.

cfunction função C que será executado no estado Lua. Essa função deve obedecer o tipo `lua_CFunction`. e deve aguardar um único parâmetro do tipo `light userdata` na pilha virtual de Lua. Ela pode retornar no máximo um único valor. Neste caso, o valor será retirado da pilha virtual e retornado através do parâmetro `resultp`. Além disso, essa função pode utilizar irrestritamente a API C de Lua.

lightuserdata parâmetro que será passado para a função `cfunction` através da pilha virtual de Lua.

resultp ponteiro para string passado como referência para receber o resultado retornado pela função C. O valor de retorno será convertido para uma string. Caso não se deseje receber o resultado da execução, deve-se passar o valor `NULL` através deste parâmetro. Em caso de erro de execução da função C, a string retornada conterá a mensagem de erro. A string contendo o resultado é alocada dinamicamente utilizando o alocador do estado `Lk`. É responsabilidade do cliente da interface desalocar essa string. Para facilitar esse processo, pode-se usar a função `lunatik_free`.

size_resultp ponteiro passado como referência para receber o tamanho da string de resultado.

lunatik_lock() bloqueia o acesso a um estado Lunatik através da função `lockf` registrada no estado. Após o retorno desta função, a exclusão mútua ao estado Lua encapsulado no estado Lunatik `Lk` é garantida (caso a função `lockf` tenha sido implementada corretamente). Essa função toma o seguinte argumento:

Lk descritor do estado Lunatik que se deseja bloquear.

lunatik_unlock() desbloqueia o acesso a um estado Lunatik através da função `unlockf` registrada no estado. Essa função toma o seguinte argumento:

Lk descritor do estado Lunatik que se deseja bloquear.

lunatik_alloc() aloca memória utilizando a função alocadora passada na criação do estado Lunatik. Em caso de sucesso, retorna um ponteiro para início do espaço de memória alocado. Caso não tenha memória disponível no sistema, retorna `NULL`. Essa função toma os seguintes argumentos:

Lk descritor do estado Lunatik.

size tamanho do espaço de memória a ser alocado.

lunatik_free() libera memória utilizando a função alocadora passada na criação do estado Lunatik. Essa função toma os seguintes argumentos:

Lk descritor do estado Lunatik.

ptr ponteiro para o início do espaço de memória que se deseja liberar.

size tamanho do espaço de memória a ser liberado.

lunatik_getluastate() obtém o estado Lua encapsulado em um estado Lunatik. Essa função toma o seguinte argumento:

Lk descritor do estado Lunatik.

lunatik_newstate() cria um novo estado Lunatik. Um estado Lunatik é um estado Lua encapsulado por um mecanismo de sincronização. Em caso de sucesso essa função retorna o novo estado criado. Caso não tenha memória suficiente disponível no sistema, ela retorna `NULL`. Essa função toma os seguintes argumentos:

lunatikL_newstate() cria um novo estado Lunatik usando o alocador de memória `kmem(9)` e o mecanismo de exclusão mútua `mutex(9)`. Essa função toma os seguintes argumentos:

km_flags flags que serão passadas para as funções e de liberação de memória, respectivamente, `kmem_alloc` e `kmem_free`.

kmutex ponteiro para o descritor de mutex. Esse descritor deve ser previamente inicializado. É também função do cliente dessa função, destruir o mutex depois do encerramento do estado Lunatik retornado.

kmutex_type tipo do mutex apontado pelo parâmetro `kmutex`. Este argumento é utilizado para determinar as funções que serão utilizadas para bloquear e desbloquear o estado Lunatik, como por exemplo funções `spin` (espera ocupada).