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.

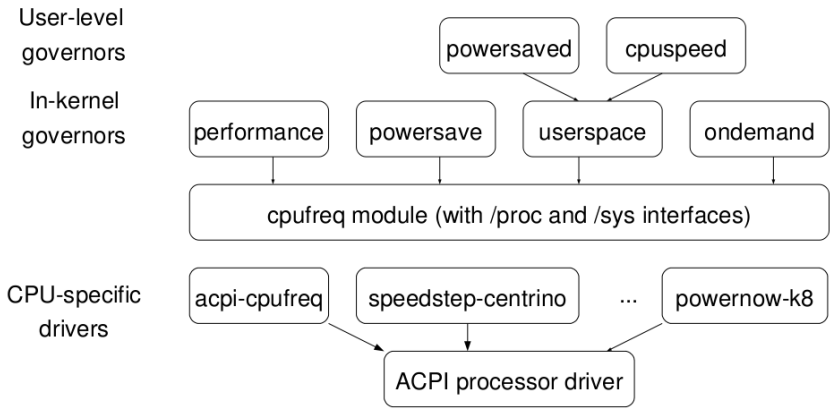

Figura 4.3: Organização do subsistema Cpufreq (31)

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