Ronan Carvalho de Resende

Gerenciamento de Recursos pelo *Kernel* do Linux Baseado em Classes

Monografia de Pós-Graduação "*Lato Sensu*" apresentada ao Departamento de Ciência da Computação para obtenção do título de Especialista em "Administração em Redes Linux"

Orientador
Prof. MSc. Joaquim Quinteiro Uchôa

Lavras
Minas Gerais - Brasil
2005

Ronan Carvalho de Resende

Gerenciamento de Recursos pelo *Kernel* do Linux Baseado em Classes

Monografia de Pós-Graduação "*Lato Sensu*" apresentada ao Departamento de Ciência da Computação para obtenção do título de Especialista em "Administração em Redes Linux"

Aprovada em *11 de Dezembro de 2005*

Prof. MSc. Herlon Ayres Camargo

Prof. MSc. Sandro Melo

Prof. MSc. Joaquim Quinteiro Uchôa
(Orientador)

Lavras
Minas Gerais - Brasil

Resumo

Este trabalho apresenta um mecanismo de Gerenciamento de Recursos pelo *kernel* do Linux Baseado em Classes como forma de controlar a utilização de recursos de CPU, memória, E/S de disco e rede. Este mecanismo possibilita o agrupamento de processos em classes com limite inferior e superior de utilização percentual de recurso.

Sumário

1 Introdução **1**

2 Gerenciamento de recursos **3**
2.1 Limitação de recursos através do PAM 3
2.2 Trabalhos correlatos 4
2.3 CKRM 5

3 Estrutura do CKRM **7**
3.1 Componentes 7
3.2 Estrutura do sistema de arquivos virtual RCFS 8
3.3 Garantias e limites para uso de CPU 10
3.4 Sintaxe para criação de regras para processos 10

4 Implementação e configuração do CKRM **13**
4.1 Comentários iniciais 13
4.2 Implementação 13
4.3 Configuração do CKRM 14
4.4 Utilização do CKRM para limitar a utilização de CPU 15
4.5 *Overhead* 16

5 Conclusão **19**
5.1 Propostas para trabalhos futuros 20

Lista de Figuras

3.1 Estrutura do CKRM 7
3.2 Árvore de diretórios RCFS 9
3.3 Sintaxe e exemplo do comando para alterar garantias 10
3.4 Sintaxe e exemplo para criação de regras 11

4.1 *Patches* 13
4.2 Aplicação dos *patches* 14
4.3 Configurações do *kernel* 14
4.4 Comando para criação e montagem do diretório rcfs 14
4.5 Comandos para criar classes e regras 15
4.6 *script* hogcpu.sh 15
4.7 `top` 16

Lista de Tabelas

2.1 Recursos que podem ser limitados pelo pam_limits 4

4.1 Limitação no uso de processador 16
4.2 *Overhead* do CKRM 17

Capítulo 1

Introdução

Ferramentas de gerenciamento de recursos são essenciais para permitir que tarefas importantes para o usuário ou o negócio tenham uma garantia mínima de recursos. Com a tendência crescente de consolidação de vários servidores em máquinas multiprocessadas, *mainframes* e utilização de *grids* e *clusters*, é necessário garantir a execução de determinadas tarefas com a prioridade desejada.

Através do gerenciamento de recursos é possível definir que tarefas consideradas importantes sejam executadas com maior prioridade que tarefas menos importantes. Isso permite que tais processos obtenham mais tempo de execução de CPU, mais banda de E/S[1] e rede e preferência na alocação de memória. Vários sistemas operacionais possuem funções de gerenciamento de recursos, exemplo: z/OS[2], HP-UX[3], AIX[4], e Solaris[5]. Nesse sentido, os mecanismos de gerenciamento de recursos disponíveis no Linux são muito limitados. Geralmente, permitem a definição de limites superiores de uso do recurso e tal restrição é feita por usuário ou por *pid*, não permitindo, por exemplo, que seja usado como critério o nome da aplicação.

Este trabalho tem o objetivo de apresentar uma nova tecnologia que tenta suprir essa deficiência do Linux: O Gerenciamento de Recursos pelo *Kernel* do Linux Baseado em Classes é um projeto que consiste de *patches* para o *kernel* do Linux, a partir da série 2.6. O nome original do projeto é *Class-based Kernel Resource Management*, abreviado como CKRM[6]. Seu objetivo é desenvolver mecanismos para o *kernel* do Linux para controlar utilização de recursos como processador, memória, E/S e banda de rede, baseado em grupos definidos pelo administrador chamados de classes.

[1] Entrada e saída neste trabalho será usado para indicar leitura e gravação em disco

[2] WLM - http://www-03.ibm.com/servers/eserver/zseries/zos/wlm/

[3] WLM - http://www.hp.com/products1/unix/operating/wlm/overview.html

[4] WLM - http://www.redbooks.ibm.com/abstracts/sg245977.html

[5] SRM - http://www.sun.com/software/resourcemgr/index.xml

[6] http://ckrm.sourceforge.net/

Este trabalho está organizado da seguinte forma: o Capítulo 2 enumera alguns mecanismos de gerência de recursos do Linux; o Capítulo 3 apresenta a estrutura do CKRM; no 4 é avaliada a utilização do CKRM para controlar o uso de CPU.

Capítulo 2

Gerenciamento de recursos

Em ambientes multitarefas e multiusuários é indispensável haver mecanismos que impeçam que um único usuário ou tarefa utilizem todos os recursos da máquina, causando lentidão ou falta de resposta aos outros usuários/tarefas. Esses mecanismos são chamados de gerenciamento de recursos ou gerenciamento de cargas e são implementados através de aumento da prioridade de tarefas mais importantes ou limitação no uso de recursos de tarefas menos importantes.

Através do gerenciamento de recursos, os prejuízos causados por usuários ou tarefas "mal comportadas" pode ser minimizado. Este capítulo descreve alguns mecanismos destinados ao gerenciamento de recursos no Linux.

2.1 Limitação de recursos através do PAM

O PAM (*Pluggable Authentication Modules*), descrito em (SAMAR; SCHEMERS, 1995), é um conjunto de bibliotecas usadas para controlar a autenticação e utilização de recursos por usuários. Nas distribuições do sistema operacional Linux que fazem uso do PAM é possível definir limites superiores para alguns recursos através do módulo `pam_limits`.

Os itens controlados estão relacionados a CPU, memória, número de *logins*, tamanho e quantidade de alocação de arquivos, ver Tabela 2.1. Não é possível restringir o uso de banda de E/S e rede através do `pam_limits`. Limites podem ser definidos com base em usuários ou grupos, mas não se aplicam a contas `uid=0 (root)` (GAFTON, 1996).

O `pam_limits` não permite, por exemplo, controlar a utilização de CPU em um dado momento, garantindo que determinado processo tenha prioridade sobre outros processos. A única limitação no uso de CPU é quanto ao tempo máximo e ao atingir esse limite a sessão e todos os processos do usuário são finalizados.

Para conseguir limitar recursos, o `pam_limits` utiliza as chamadas de sistema `getrlimit(2)`, `setrlimit(2)`, e `getrusage(2)`, que também são

Tabela 2.1: Recursos que podem ser limitados pelo `pam_limits`

Recurso	Descrição
$core$	tamanho máximo para arquivos `core` (KB)
$data$	tamanho máximo do segmento de dados de um processo na memória
$fsize$	tamanho máximo para arquivos que forem criados
$memlock$	tamanho máximo de memória que um processo pode bloquear na memória física (KB)
$nofile$	quantidade máxima de arquivos abertos de cada vez
rss	tamanho máximo de memória que um processo pode manter na memória física (KB)
$stack$	tamanho máximo da pilha (KB)
cpu	tempo máximo de uso de CPU (em minutos)
$nproc$	quantidade máxima de processos disponíveis para um único usuário
as	limite para o espaço de endereçamento
$maxlogins$	quantidade máxima de *logins* para este usuário
$priority$	a prioridade com que os processos deste usuário serão executados

utilizadas pelo comando de *shell* `ulimit`. O `ulimit` é uma alternativa para limitar recursos, mas também é bem restrita e se aplica ao recursos apresentados na Tabela 2.1, exceto quantidade máxima de logins e prioridade dos processos do usuário.

2.2 Trabalhos correlatos

Em (STARKE; MAZIERO; JAMHOUR, 2004) é apresentado um protótipo capaz de controlar dinamicamente a alocação de tempo de processador entre vários processos. Os autores não tornaram os códigos disponíveis para implementação e avaliação.

O Linux Soft Real-Time (Linux-SRT)(CHILDS; INGRAM, 2001) foi desenvolvido para melhorar o tempo de resposta para aplicações de tempo real. O Linux-SRT permite definir percentagens e limites de utilização de CPU para cada processo. Disponível como um *patch* para o *kernel* da série 2.2, o projeto está descontinuado e a última atualização foi em maio de 2000.

Cap Processor Usage (CPU)(GOLAB, 2002) é um *patch* para o *kernel* da série 2.4 (até 2.4.18), apenas para a arquitetura i386. O *patch* limita a percentagem máxima que processos podem utilizar do processador, mas não assegura uma alocação mínima. Limites são definidos pelo *pid* da *task*, o que não possibilita o agrupamento e a definição de limites com base em usuário ou grupo e dificulta a imposição de limites por nome do processo.

O ARMTech é um gerenciador de recursos para a distribuição SuSE SLES8 e SLES9[1] vendido pela empresa Aurema e suportado oficialmente pela Novell.

[1]SLES - *SuSE Linux Enterprise Server*, versão destinada à empresas

De acordo com (AUREMA PTY LIMITED, 2005) é possível definir limite inferior e superior para o percentual de utilização do processador. Regras podem ser criadas para usuário, grupo ou aplicação.

2.3 CKRM

Um mecanismo mais completo para o gerenciamento de recursos no Linux é o CKRM, acrônimo de *Class-based Kernel Resource Management*. A ferramenta consiste de *patches* para o *kernel* a partir da série 2.6 e é aplicável a todas as plataformas. Entre outras características, o que o torna mais interessante é a possibilidade de controlar a utilização de CPU, memória, banda de E/S e rede.

O CKRM permite a classificação automática de processos, estabelecendo critérios baseados em *uid*, *gid*, nome ou caminho do processo. Soquetes podem ser classificados de acordo com endereço/porta de origem ou destino. Os processos ou soquetes são agrupados em classes e a cada classe são atribuídos limites e garantias para utilização de recurso.

O CKRM pode evitar que um processo ao entrar em *loop* de consumo de CPU ou memória impeça que tarefas essenciais sejam executadas. Basta criar regras para as aplicações que de vez em quando elevam de forma anormal o consumo de recursos e definir: limites menores que 100 para a classe dessas aplicações e garantias maiores que 0 para todas as classes. Assim, tais processos irão obter 100% dos recursos caso estes estejam livres e quando processos de outras classes necessitarem de recursos, estes serão assegurados de acordo com a garantia da classe. Dessa forma, o CKRM também contribui para evitar ataques de negação de serviço.

Processos de *backup*, por exemplo, podem ser classificados em classes que tenham baixa prioridade de E/S, pois dessa forma não prejudicam requisições de E/S de outros processos. A prioridade é definida como um limite inferior e caso não haja contenção pelo uso do recurso, o processo usa mais do que o limite.

O CKRM fornece informação de quanto foi utilizado por cada classe, o que permite ao administrador conhecer os serviços que mais consomem recursos, fornece dados para auxiliar no planejamento de capacidade e possibilita a contabilização para fins de definir o custo de determinado serviço.

Dadas as inúmeras vantagens do CKRM sobre suas alternativas, ele é abordado mais detalhadamente nos capítulos a seguir.

Capítulo 3

Estrutura do CKRM

3.1 Componentes

A estrutura do CKRM é formada pelos componentes apresentados na Figura 3.1.

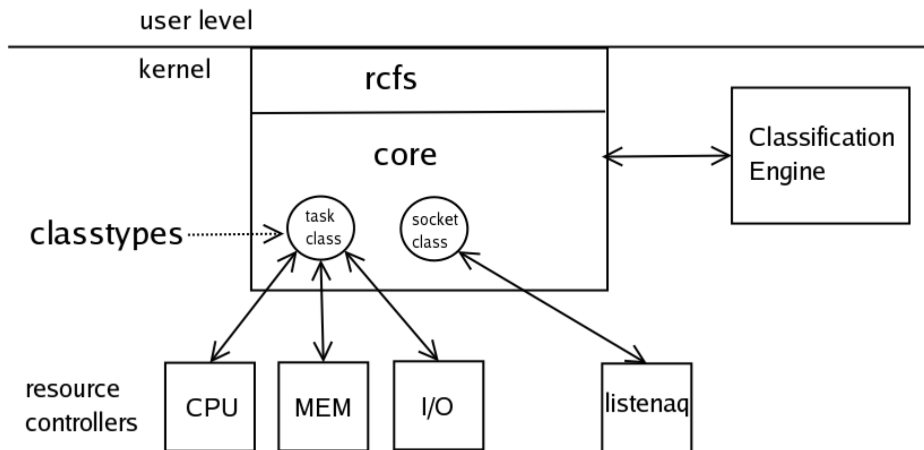

Figura 3.1: Estrutura do CKRM

Os principais componentes do CKRM são apresentados a seguir:

Núcleo (*Core*): O núcleo é a entidade básica do CKRM e serve para interação entre os outros componentes. É um *patch* que define uma API[1] composta de chamadas de sistema e um sistema de arquivos virtual chamado RCFS (*resource control file system*).

[1]*Application Programming Interface* é um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades.

É o núcleo que permite a criação e o gerenciamento de classes. Uma classe é um grupo de objetos associados a garantias, limites e estatísticas de utilização. O CKRM atualmente define 2 tipos de classes: *taskclass* e *socketclass*, que agrupam respectivamente tarefas e soquetes. Sub-classes podem ser definidas e estas dividem os recursos alocados para a classe a que pertencem.

Mecanismo de classificação (CE - *Classification Engine*): Possibilita classificação automática de tarefas e soquetes em classes, de acordo com regras definidas. O CKRM possui um mecanismo de classificação baseado em regras, chamado de RBCE (rule-based classification engine). Todo objeto gerenciado pelo CKRM é associado a uma classe. Se algum objeto não satisfaz nenhuma regra, ele pertencerá a classe padrão do seu tipo (*taskclass* ou *socketclass*). O mecanismo de classificação é opcional. Objetos podem ser manualmente associados às classes.

Em chamadas de sistema como `fork()`, `exec()` e `listen()`, quando atributos de objetos são alterados, o núcleo pergunta ao mecanismo de classificação, caso exista, qual deve ser a classe do objeto.

Controlador/Gerenciador de recursos (RC - *Resource controller*): A cada recurso pode ser associado um controlador. O CKRM atualmente provê controladores para CPU, memória, E/S de disco e rede (*inbound connections*).

Sistema de arquivos de controle de recursos (RCFS): O RCFS é um sistema de arquivos virtual e funciona como principal interface entre usuário e *kernel* para o CKRM. Após montado, o sistema de arquivos provê uma hierarquia de diretórios e arquivos que podem ser manipulados através de operações de arquivos tais como `open()`, `close()`, `read()`, `write()`, `mkdir()`, `rmdir()` e `unlink()`.

3.2 Estrutura do sistema de arquivos virtual RCFS

O sistema de arquivos virtual de controle de recursos (RCFS), após montado no diretório `/rcfs` e após a criação da classe `c1` e da regra `r1`, apresenta os arquivos/diretórios exibidos na Figura 3.2.

Segue-se a função dos arquivos/diretórios mais importantes da árvore de diretórios RCFS. Para mais detalhes ver (SEETHARAMAN; NAGAR; KASHYAP, 2004).

`ce`: Diretório que é interface para o mecanismo de classificação.

`rbce_state`: Determina se o RBCE está ativo (1) ou inativo (0). O status do RBCE pode ser alterado escrevendo 1 ou 0 no arquivo.

```
/rcfs/
|-- ce
|   |-- rbce_state
|   |-- rbce_tag
|   `-- rules
|       `-- r1
|-- socketclass
|   |-- members
|   `-- reclassify
`-- taskclass
    |-- c1
    |   |-- members
    |   |-- shares
    |   `-- stats
    |-- config
    |-- members
    |-- reclassify
    |-- shares
    `-- stats
```

Figura 3.2: Árvore de diretórios RCFS

`rules`: Diretório onde são criadas as regras para classificação. Cada regra do RBCE é representada por um arquivo. A sintaxe para criação de regras para classificação automática de processos é apresentada na Seção 3.4.

`r1`: Regra `r1` criada.

`taskclass`: Diretório onde são criadas as classes de processos. Cada classe é um sub-diretório. A criação de classes é feita através do comando `mkdir`.

`reclassify`: Escrever o `pid` de um processo neste arquivo faz com que as regras sejam examinadas para reclassificação do processo. Essa classificação é feita automaticamente quando o processo é criado. Quando a regra é criada ou alterada e o processo já está ativo, escrever o processo no `reclassify` força a reclassificação do processo.

`c1`: Classe `c1` criada com comando `mkdir`.

`shares`: Arquivo com valores de garantia e limite (ver Seção 3.3) para utilização do recurso.

`stats`: Arquivo com estatísticas de utilização.

`members`: Arquivo que contém a lista dos processos de cada classe. Processos podem ser classificados automaticamente pelo RBCE ou manualmente, escrevendo o processo no arquivo `members` da classe.

3.3 Garantias e limites para uso de CPU

Como comentado na Seção 3.2, no arquivo *shares* são especificados os valores de garantia e limite para utilização dos recursos.

Garantia (*guarantee*) é a quantidade percentual mínima de recursos que os processos de uma classe obtêm, caso necessitem. Caso não exista demanda a classe pode utilizar menos do que sua garantia. Quando há recurso disponível, a garantia funciona como um peso relativo da classe em relação às outras e o recurso livre é redistribuído de acordo com esse peso. O valor padrão é `0` e significa que não há nenhuma garantia e a classe pode não obter o recurso, caso não esteja disponível.

Exemplo: uma classe com garantia 5 e outra classe tem garantia 10. Se a CPU estiver disponível 60% do tempo, esse tempo será distribuído proporcionalmente: 20% e 40%, respectivamente.

A soma das garantias de todas as classes não pode ser maior que 100. Quando um processo não satisfaz nenhuma regra de classificação e nem foi classificado manualmente, ele pertencerá à classe padrão. A garantia da classe padrão é 100 menos a soma das garantias das outras classes. Portanto, é interessante assegurar que essa soma seja menor que 100 para que a classe padrão tenha uma garantia diferente de 0, porque nessa classe podem estar diversos processos do kernel e do sistema.

Limite (*limit*) é a quantidade máxima percentual de recursos que uma classe pode obter quando há contenção pelo uso do recurso. Quando o recurso CPU estiver disponível a classe pode usar além do limite, ou seja, para utilização de CPU o limite é maleável (*soft limit*). O valor padrão é `100` e significa que não há limite e a classe pode usar todo o recurso livre.

A Figura 3.3 apresenta a sintaxe e exemplo de alteração da garantia para a sub-classe `1` da classe `c1`.

```
sintaxe: echo res=<resource>,[guarantee=<número>,]\
              [limit=<número>,]\
              > /rcfs/taskclass/<classe>/shares

# echo res=cpu,guarantee=5 > /rcfs/taskclass/c1/1/shares
# cat /rcfs/taskclass/c1/1/shares
res=cpu,guarantee=5,limit=100,total_guarantee=100,max_limit=100
```

Figura 3.3: Sintaxe e exemplo do comando para alterar garantias

3.4 Sintaxe para criação de regras para processos

Regras para classificação automática de processos são criadas através de adição de linhas em arquivos do diretório `/rcfs/ce/rules/`. A Figura 3.4 apresenta a

sintaxe para criação de regras de classificação, bem como exemplos de uso. Nesse caso, tem-se os seguintes parâmetros:

critério: <*id> <OP> número, sendo que <OP>={>,<,=} <*id> = {uid, euid, gid, egid}
cmd = nome do comando
pathname = caminho completo do comando
args = argumentos
apptag = *tag* da aplicação.

ordem: `order=<number>`, ordem na qual a regra é examinada. A regra de ordem 1 é a primeira a ser checada. Assim que o critério de uma regra é satisfeito, a classe correspondente a esta regra é retornada para o Núcleo. Portanto a ordem em que são examinadas as regras é importante. Parâmetro opcional e quando não especificado, é atribuído automaticamente em ordem crescente de 10 em 10.

estado: `state=1 ou 0`, estado ativo(1) ou inativo(0) da regra. Permite ativar ou desativar uma regra. Também é opcional e quando não informado, é assumido valor 1.

classe: `class=/rcfs/taskclass/<classe>`, classe do processo caso o critério seja satisfeito.

```
sintaxe: echo critério,ordem,estado,classe > /rcfs/ce/rules/<regra>

# echo cmd=hogcpu,class=/rcfs/taskclass/limitada > /rcfs/ce/rules/r1
# echo "uid>0,class=/rcfs/taskclass/limitada"    > /rcfs/ce/rules/r2
# echo "gid=104,class=/rcfs/taskclass/classeA"   > /rcfs/ce/rules/r3
```

Figura 3.4: Sintaxe e exemplo para criação de regras

Este Capítulo apresentou a estrutura do CKRM, a forma de definição de garantias/limites e de criação de regras. No Capítulo 4 será apresentado a implementação e configuração do CRKM para controlar a utilização de CPU.

Capítulo 4

Implementação e configuração do CKRM

4.1 Comentários iniciais

Neste Capítulo será apresentado como implementar e configurar o CKRM para controlar a utilização de CPU. Na opinião deste autor, este é um dos controles mais difíceis e interessantes. Para implementação do CKRM foi utilizada uma máquina AMD 1.1 GHz, 256 MB de memória RAM, distribuição Linux/Slackware 10.0, *kernel* 2.6.12

O CKRM está disponível na forma de *patches* para o *kernel* do Linux. Os *patches* para o CKRM podem ser aplicados de forma independente, exceto para o núcleo, que é obrigatório. Para controlar a utilização de CPU, por exemplo, pode ser aplicado apenas o *patch* que implementa esta funcionalidade.

4.2 Implementação

Os *patches* que devem ser aplicados no *kernel* 2.6.12 para controlar a utilização de CPU são apresentados na Figura 4.1 e podem ser obtidos na página do projeto CKRM[1].

```
ckrm_e18_2612_single.patch
cpu.ckrm-e18.v10.patch
```

Figura 4.1: *Patches*

Os procedimentos para aplicar os *patches* são apresentados na Figura 4.2.

[1] http://sourceforge.net/project/showfiles.php?group_id=85838

```
# cd /usr/src/linux
# patch -p1 < /ckrm/ckrm_e18_2612_single.patch
# patch -p1 < /ckrm/cpu.ckrm-e18.v10.patch
```

Figura 4.2: Aplicação dos *patches*

É necessário configurar e compilar o *kernel* com as opções da Figura 4.3. Detalhes sobre os procedimentos necessários para compilar o *kernel* podem ser obtidos em (WARD, 1997).

```
General Setup--> Class Based Kernel Resource Management
     [*] Class Based Kernel Resource Management Core
     <*>   Resource Class File System (User API)
     [*]     Class Manager for Task Groups
     [*]       CKRM CPU scheduler (NEW)
     [ ]         Turn on at boot time (NEW)
     [ ]     Class Manager for socket groups
     < > Number of Tasks Resource Manager
     <M> Classification Engine
     <M>   Rule-based Classification Engine (RBCE)
```

Figura 4.3: Configurações do *kernel*

4.3 Configuração do CKRM

Após carregar o *kernel* compilado com o CKRM, é necessário criar e montar o sistema de arquivos RCFS, Figura 4.4.

```
# mkdir /rcfs
# mount -t rcfs rcfs /rcfs
```

Figura 4.4: Comando para criação e montagem do diretório rcfs

A leitura e alteração de valores do /rcfs funciona de forma semelhante ao /proc. Valores podem ser lidos com o comando `cat <arquivo>` e alterados com `echo > <arquivo>`. Classes são representadas por diretórios e podem ser criadas ou removidas com os comandos `mkdir` e `rmdir`, respectivamente.

Após a montagem do sistema de arquivos RCFS, foram executados os comandos listados na Figura 4.5 para a criação das classes, definição de garantias e criação das regras.

```
# insmod  /lib/modules/2.6.12/kernel/kernel/ckrm/rbce/rbce.ko
# echo res=cpu,mode=enabled > /rcfs/taskclass/config

****  Criação das classes
# mkdir /rcfs/taskclass/classeA
# mkdir /rcfs/taskclass/classeB
# mkdir /rcfs/taskclass/classeC
# mkdir /rcfs/taskclass/classeD

****  Definição de garantias e limites
# echo res=cpu,guarantee=40,limit=90 > /rcfs/taskclass/classeA/shares
# echo res=cpu,guarantee=20,limit=60 > /rcfs/taskclass/classeB/shares
# echo res=cpu,guarantee=10,limit=20 > /rcfs/taskclass/classeC/shares
# echo res=cpu,guarantee=5,limit=40 > /rcfs/taskclass/classeD/shares

****  Criação das regras
# echo uid=1003,class=/rcfs/taskclass/classeA > /rcfs/ce/rules/regra1
# echo uid=1004,class=/rcfs/taskclass/classeB > /rcfs/ce/rules/regra2
# echo uid=1005,class=/rcfs/taskclass/classeC > /rcfs/ce/rules/regra3
# echo uid=1006,class=/rcfs/taskclass/classeD > /rcfs/ce/rules/regra4
```

Figura 4.5: Comandos para criar classes e regras

4.4 Utilização do CKRM para limitar a utilização de CPU

Para verificar se o CKRM cumpre a função de garantir o percentual de utilização para cada classe, foram iniciadas 4 instâncias do programa `hogcpu.sh` (Figura 4.6), cuja finalidade é usar todo o tempo de processador disponível. Cada instância do programa foi iniciada pelos usuários `usera, userb, userc e userd` com `uids 1003, 1004, 1005 e 1006` respectivamente.

```
#!/bin/bash
i=1
while true ; do
  let "i= $i + 1"
  let "i= $i - 1"
done
```

Figura 4.6: *script* hogcpu.sh

Para observar a utilização de processador foi utilizado o programa `top`, Figura 4.7. A Tabela 4.1 apresenta um resumo das informações do teste efetuado. Os valores da coluna `Peso` foram obtidos dividindo a garantia de cada classe pela soma de todas as garantias. O valor de `Utilização` é a divisão do tempo que cada processo gastou de CPU pelo tempo de todos os processos. Comparando os valores dessas duas colunas, observa-se que são bem próximos, o que significa que o CKRM efetivamente garantiu a alocação de tempo de CPU para cada classe. Embora os 4 processos estivessem demandando 100% de processador cada, o tempo obtido foi de acordo com a prioridade estabelecida. Durante o teste, o sistema não

```
top - 00:57:14 up 12 min,  6 users,  load average: 3.99, 3.34, 1.76
Tasks:  61 total,   5 running,  56 sleeping,   0 stopped,   0 zombie
Cpu(s): 98.3% us,  1.7% sy,  0.0% ni, 0.0% id, 0.0% wa, 0.0% hi, 0.0% si
Mem:    482640k total,    63800k used,   418840k free,     7136k buffers
Swap:        0k total,        0k used,        0k free,    31404k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3188 usera     25   0  4328 1012  888 R 44.7  0.2   4:03.56 hogcpu.sh
 3190 userb     25   0  4332 1016  888 R 33.1  0.2   2:44.47 hogcpu.sh
 3192 userc     25   0  4332 1016  888 R 14.9  0.2   1:21.48 hogcpu.sh
 3194 userd     25   0  4328 1016  888 R  7.0  0.2   0:40.60 hogcpu.sh
```

Figura 4.7: `top`

deixou de responder porque seus processos pertenciam a classe padrão que tinha uma garantia de 35 (100 - 65). Caso não houvesse o CKRM com a configuração utilizada, o sistema travaria.

Tabela 4.1: Limitação no uso de processador

Usuário	*uid*	Classe	Garantia	Peso	Tempo	Utilização (%)
usera	1003	classeA	30	46.15	04:03.56	45.95
userb	1004	classeB	20	30.77	02:44.47	31.03
userc	1005	classeC	10	15.38	01:21.48	15.37
userd	1006	classeD	5	7.69	00:40.60	7.66
Total			65	100	08:50.11	100

4.5 *Overhead*

Para medir o *overhead* causado pelo CKRM foi utilizado o programa de *benchmark* de sistemas `unixbench-4.1.0`[2] em uma máquina equipada com Pentium II 300 MHz, 320 MB de memória RAM, distribuição Linux/Slackware 10.0, *kernel* 2.6.12. Foram executados testes com 4 configurações diferentes:

***kernel* sem CKRM:** os valores obtidos nesse teste serviram como referência para comparar o *overhead* introduzido nos outros testes.

***kernel* com CKRM:** *kernel* compilado com os *patches* do CKRM, mas sem nenhuma regra ou classe criada e nem o RCFS montado.

com CKRM e sem regras: situação anterior, mas com uma classe definida com garantia 5 e limite 20. Não foram criadas regras e os processos de *benchmark* foram manualmente associados à classe.

[2] http://www.tux.org/pub/tux/niemi/unixbench/

com CKRM e com regras: classe definida com os mesmos valores do teste anterior e com uma regra que classifica os processos do *benchmark* associando-os à classe.

Para cada configuração, o programa de *benchmark* foi executado 3 vezes com a opção `system`, que faz vários testes de performance de sistema e apresenta os índices obtidos. Foi considerada a média dos valores das 3 execuções. Os resultados estão apresentados na Tabela 4.2, onde valores negativos significam índices piores e positivos índices melhores em comparação com os testes realizados com o *kernel* sem o CKRM.

Tabela 4.2: *Overhead* do CKRM

Benchmark	CKRM	Classif. Manual	Classif. por regra
Execl Throughput	2.7%	2.8%	2.2%
File Copy 1024 bufsize 2000 maxblocks	-2.6%	-1.2%	-3.7%
File Copy 256 bufsize 500 maxblocks	-5.6%	-5.3%	-5.7%
File Copy 4096 bufsize 8000 maxblocks	-1.5%	-2.9%	-1.9%
Pipe Throughput	-1.1%	0.8%	-1.9%
Pipe-based Context Switching	3.3%	4.5%	-30.9%
Process Creation	-5.4%	-5.1%	-4.4%
Shell Scripts (8 concurrent)	0.0%	0.0%	0.0%
System Call Overhead	-2.9%	-2.8%	-2.2%

De acordo com os testes realizados, o *overhead* gerado pelo CKRM ficou abaixo de 6% em todos os casos, exceto um, onde o *overhead* foi de 30,9%. O *benchmark Pipe-based Context Switching* consiste de medir o número de vezes que 2 processos podem trocar entre si um inteiro crescente (LOSCOCCO; SMALLEY, 2001). A perda de performance é percebida apenas quando se altera o estado do controlador de CPU de inativo para ativo. Questionados sobre esse *overhead* alto, Matt Helsley e Chandra Seetharaman, autores do projeto CKRM, recomendaram testar uma nova versão, o que não pôde ser feito no tempo de confecção deste trabalho.

Capítulo 5

Conclusão

Há uma discussão se o CKRM deve ser incluído diretamente no *kernel* oficial. O principal argumento contra a inclusão é apresentado em (CORBET, 2005):

> O CKRM estabelece muitos ganchos em partes críticas do *kernel* para monitorar a transição de processos e garantir limites de utilização. Portanto, o CKRM precisa ser testado por grande número de pessoas antes, para garantir que o código não introduz erros no *kernel*.

O fato de não estar na árvore oficial do *kernel* faz com que a cada versão nova do *kernel* os *patches* para CKRM tenham grande possibilidade de deixar de funcionar. Então, os *patches* são específicos para uma versão. Até mesmo *scripts* e comandos utilizados para configurar o CKRM escrevendo no sistema de arquivos RCFS podem não funcionar ao mudar a versão dos *patches*, pois ocorrem alterações na sintaxe dos comandos e nos diretórios. Por este motivo, o autor deste trabalho teve dificuldades na implementação. A pouca documentação disponível sobre a implementação e configuração do CKRM apresentavam trechos incorretos, que não levavam em consideração as alterações de versão.

Porém, os benefícios apresentados superam eventuais desvantagens, porque o CKRM é o mecanismo mais eficiente de gerenciamento de recursos no Linux que este autor tem conhecimento. Nos testes apresentados na Seção 4.4 ele foi capaz de garantir a alocação de tempo de CPU de acordo com os parâmetros especificados. É importante observar, inclusive, que a distribuição SuSE, incluiu o CKRM no *kernel* instalado com sua versão SLES9.

De acordo com (FRANKE *et al.*, 2004), o gerenciamento de recursos tem importância cada vez maior nos sistemas computacionais modernos. Com a consolidação de servidores, onde aplicações que executam em máquinas separadas passam a rodar na mesma máquina, é desejável evitar que aplicações pouco importantes para o negócio atrapalhem outras de alta prioridade. Em servidores *web*, por exemplo, a transação de um cliente que está fazendo pagamento deve ter preferência sobre

a transação de cliente que está apenas navegando. Priorizar determinado tipo de serviço, limitando outro, pode ajudar a prevenir ataques de negação de serviço.

5.1 Propostas para trabalhos futuros

Devido à pouca documentação (nenhuma em português) sobre o CKRM e a inexistência de todos os controladores de recursos, este trabalho concentrou-se na avaliação do controle do uso de CPU. Como primeira sugestão para trabalhos futuros, a avaliação dos outros controladores: memória, E/S para disco e rede. Outras sugestões:

***scripts*:** desenvolvimento de *scripts* para facilitar o uso do CKRM. Por exemplo: como saber a garantia e limite de todas as classes?

***patches*:** desenvolvimento de *patches* para programas como o `ps` e o `top` para que exibam uma coluna com a classe dos processos. Atualmente, não é fácil saber a classe de um processo. Para isso, é preciso saber o `pid` do processo desejado e pesquisar os arquivos `members` de todas as classes.

interface gráfica: desenvolvimento de interface gráfica para visualização e alteração de parâmetros do CKRM.

controladores: contribuição no desenvolvimento de gerenciadores de recursos.

Referências Bibliográficas

AUREMA PTY LIMITED. *ARMTech User Guide for SuSE Linux Enterprise Server 8 & 9*. [S.l.], 19 set. 2005. Disponível em: <http://www.aurema.com-/resources/assets/ARMUG_suse.zip>. Acesso em: 10/10/2005.

CHILDS, S.; INGRAM, D. The linux-srt integrated multimedia operating system: Bringing qos to the desktop. In: *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*. Washington, DC, USA: IEEE Computer Society, 2001. p. 135.

CORBET, J. *Is CKRM worth it?* 2005. Disponível em: <http://lwn.net/Articles-/145135/>. Acesso em: 10/10/2005.

FRANKE, H.; SEETHARAMAN, C.; NAGAR, S.; KASHYAP, V. *Advanced Workload Management Support for Linux.* 2004. Disponível em: <http://ckrm-.sourceforge.net/downloads/ckrm-linuxtag04-paper.pdf>.

GAFTON, C. *The Linux-PAM System Administrators' Guide*. Nov 1996. Disponível em: <http://www.kernel.org/pub/linux/libs/pam/Linux-PAM-html-/pam-6.html>. Acesso em: 10/10/2005.

GOLAB, K. *CPU - Cap Processor Usage*. 2002. Disponível em: <http://www-.tlstechnologies.com/CPU/cpu-intro.html>. Acesso em: 02/10/2005.

LOSCOCCO, P.; SMALLEY, S. Integrating flexible support for security policies into the linux operating system. In: . Berkeley, CA, USA: USENIX Association, 2001. p. 29–42. ISBN 1-880446-10-3. Disponível em: <http:-//www.nsa.gov/selinux/papers/freenix01/freenix01.html>. Acesso em: 10/10/2005.

SAMAR, V.; SCHEMERS, R. J. *Unified login with pluggable authentication modules (PAM)*. [S.l.], Oct 1995. Disponível em: <http://www.opengroup.org-/tech/rfc/mirror-rfc/rfc86.0.txt>. Acesso em: 10/10/2005.

SEETHARAMAN, C.; NAGAR, S.; KASHYAP, V. *CKRM Design.* Feb 2004. Disponível em: <http://ckrm.sourceforge.net/ckrm_design.htm>. Acesso em: 10/10/2005.

STARKE, M.; MAZIERO, C.; JAMHOUR, E. Controle dinâmico de recursos em sistemas operacionais. *Anais do WSO 2004*, 2004. Disponível em: <http://www-.ppgia.pucpr.br/pesquisa/sisdist/papers/2004-wso-starke-maziero-jamhour.pdf>. Acesso em: 10/10/2005.

WARD, B. *The Linux Kernel HOWTO*. May 1997. Disponível em: <http://www-.tldp.org/HOWTO/Kernel-HOWTO/>. Acesso em: 10/10/2005.