Márcio Roberto Starke

Controle Dinâmico de Recursos em Sistemas Operacionais

Dissertação submetida ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito final à obtenção do título de Mestre em Informática Aplicada.

Curitiba

2005

Márcio Roberto Starke

Controle Dinâmico de Recursos em Sistemas Operacionais

Dissertação submetida ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito final à obtenção do título de Mestre em Informática Aplicada.

Área de Concentração: Metodologia e Técnicas de Computação

Orientador: Prof. Dr. Carlos A. Maziero

Curitiba

2005

Starke, Márcio Roberto
S795c Controle dinâmico de recursos em sistemas operacionais / Márcio Roberto
2005 Starke ; orientador, Carlos Alberto Maziero. 2005.
97 f. : il. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Paraná, Curitiba, 2005
Inclui bibliografia

1. Sistemas operacionais (Computadores). 2. Interfaces (Computadores). 4. Programação (Computadores). I. Maziero, Carlos Alberto. II. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada. III. Título.

CDD 20. ed. 005.42
004.616
005.26

A minha amada esposa e aos meus queridos pais.

O tempo é o mestre de tudo.

(Sófocles)

// *Agradecimentos*

Primeiramente a Deus, autor da vida e que sempre me guia por caminhos seguros.

Aos meus pais Paulo e Elizete que me zelaram, educaram-me e fizeram de mim o que sou hoje.

A minha esposa Sílvia que sempre me apoiou para que juntos sempre estivéssemos a subir.

Ao meu orientador Carlos Maziero pela orientação, oportunidades e compreensão.

Aos amigos Emerson e Fabiano pela convivênvia e pela eterna amizade.

E a CAPES pelo apoio financeiro tão necessário para realização desse trabalho.

Resumo

Uma das principais funções dos sistemas operacionais é a de ser gerente dos recursos do computador. Em um sistema operacional de rede não é diferente, porém, a quantidade de recursos que devem ser gerenciados é maior. Mesmo assim, os sistemas operacionais de mercado fornecem somente mecanismos limitados de gerenciamento de recursos, onde não é possível impor restrições ou privilégios de acesso aos recursos de um processo durante a sua execução. Também é notório que a gestão dos recursos nestes sistemas operacionais é feita sobre parâmetros de processos, não permitindo o ajuste do uso dos recursos por usuários, grupo de usuários ou conjunto de processos.

Esta dissertação objetiva a descrição de um modelo flexível de gestão dos recursos computacionais, que permita definir e alterar a quantidade alocada de recursos para um processo específico ou um grupo de processos relacionados por algum parâmetro, como usuários, grupo de usuários ou processos de uma aplicação, sendo que essas definições são aplicadas instantaneamente a partir do momento de sua criação.

Os recursos que este modelo propõe a gerenciar são o uso do processador, de memória, espaço em disco, banda de rede e de disco e recursos internos do sistema operacional. Para validar esta proposta foi implementado um protótipo de gerenciamento dinâmico de utilização do processador, capaz de privilegiar ou restringir seu uso de acordo com especificações feitas pelo administrador do sistema.

Este modelo pode prover uma melhor distribuição dos recursos entre os processos em execução, bem como, pode melhor adequar o sistema computacional às necessidades da organização que o mantém.

Esta proposta visa prover uma interface de programação (API) que pode ser utilizada por outros desenvolvedores que necessitem implementar um sistema operacional com suporte a qualidade de serviço (QoS); por administradores que desejam uma alternativa mais sofisticada no controle dos recursos do sistema; por aplicativos desenvolvidos com garantia de acesso aos recursos para sua execução apropriada; e por sistemas protegidos contra invasão e contra falhas como método de resposta aos ataques.

Palavras-Chave: *Controle Dinâmico, Gestão de Recursos, Sistemas Operacionais.*

Abstract

One of the main functions of the operating systems is to manage the computer resources. A network operating system has the same duty, however, the quantity of resources that should be manage is bigger. Even so, COTS operating systems supply only limited mechanisms of resources management, where is not possible impose contraints or privileges of access to the resources used by a process in execution. Also it is notorious that the management of the resources in these operating systems is done by parameters of one process, not permitting adjust the use of the resources by users, users groups or processes groups.

This dissertation objective is the description of a flexible model of management of the computer resources, that permit to define and alter the quantity of allocated resources for a specific process or a processes group related by some parameter, as users, users group or process of an application. That definitions are applied instantly from the moment of its creation.

The resources that this model proposes to manage are the use of the processor, of memory, disk space, network and disk band and internal resources of the operating system. For the validation of this proposal was implemented a prototype of dynamic management of processor utilization, capable of privilege or restrain the CPU use in agreement with the specifications made by the system administrator.

This model can supply a better distribution of the resources among the processes in execution, and it can better adapt the computer system to the needs of the organization that maintain it, as well.

This proposal is going to supply a programming interface (API) that can be utilized by others developers that are going to implement an operating system with quality of service (QoS); by administrators that desire a more sophisticated alternative in the control of the resources of the system; by applications developed with guarantee of access to the resources for their appropriate execution; and by systems protected against invasion and fail as a method of attack response.

Keywords: *Dynamic Control, Resource Management, Operational Systems.*

Sumário

1 Introdução **1**

1.1 Contexto geral 1

1.2 Motivação 2

1.2.1 Objetivo geral 4

1.2.2 Objetivos específicos 4

1.3 Estrutura do texto 5

2 Gestão de recursos **6**

2.1 Introdução 6

2.2 Recursos de um sistema de computação 7

2.2.1 Gestão do processador 7

2.2.2 Gerência de memória 14

2.2.3 Gestão de disco 18

2.2.4 Gestão de rede 20

2.2.5 Recursos internos 22

2.2.6 Classificação dos recursos 23

2.3 Conclusões 24

3 Mecanismos avançados de gestão de recursos **25**

3.1 Introdução 25

3.2 Escalonamento baseado em prioridades 25

3.3 Mecanismos avançados de gestão de recursos 30

3.3.1 Limits 30

3.3.2 Class-Based Kernel Resource Management — CKRM 32

3.3.3 Linux-SRT 33

3.3.4 Dynamic Soft Real-Time CPU Scheduler 34

3.3.5 Cap Processor Usage 35

3.4 Conclusões 36

4 Um modelo de gestão de recursos **37**

4.1 Hierarquia do uso dos recursos 37

4.2 O controle dinâmico do uso dos recursos 38

4.2.1 Processador 41

4.2.2 Memória 44

4.2.3 Disco 47

4.2.4 Rede 50

4.2.5 Recursos internos 52

4.3 Implementação do modelo 53

4.4 Conclusões 55

5 Controle dinâmico de processador **57**

5.1 Descrição do problema 57

5.2 O escalonador de processos no Linux 58

5.3 O algoritmo de escalonamento proposto 61

5.4 Descrição da implementação 62

5.5 Avaliação do protótipo 68

5.5.1 O desempenho e custo da implementação 72

5.6 Conclusões 73

6 Considerações finais **74**

6.1 Resumo do trabalho realizado 74

6.2 Principais contribuições 75

6.3 Limitações 75

6.4 Trabalhos correlatos 77

6.5 Perspectivas e trabalhos futuros 79

A Código fonte do protótipo **85**

B Código fonte do monitor de uso da CPU **94**

C Código fonte do manipulador do protótipo **96**

C.1 resourcelimits.h 96

C.2 chlimits.c 96

Lista de Figuras

2.1 Diagrama de estado dos processos. 8

2.2 Ordem de execução de processos pelo algoritmo SJF. 11

2.3 Relação entre endereços virtuais e endereços físicos. 16

3.1 Uso do processador por N processos no sistema operacional Windows XP . 28

3.2 Projeto de implementação do CKRM. 33

4.1 Aplicação de limite inferior (esquerda) e superior (direita) no uso do processador. 39

4.2 A granularidade do acesso aos recursos. 40

4.3 Utilização do processador por P1 durante N ciclos. 42

4.4 Compartilhamento da banda de rede através de regras. 51

5.1 Fluxograma do controle dinâmico de recursos. 67

5.2 Uso da CPU por P1, com N processos disputando a CPU. 69

5.3 Uso da CPU por P1 e P2 pertencentes a uma regra limite superior. 70

5.4 Uso da CPU por P1, com N processos disputando a CPU. 70

5.5 Uso da CPU por P1 e P2 pertencentes a uma regra limite inferior. 71

5.6 Acréscimo de tempo de execução imposto pelo mecanismo de aplicação das regras. 72

Lista de Tabelas

3.1 Uso do processador por N processos com diferentes prioridades durante a inserção de um novo processo. 27
3.2 Diferenças entre os modelos alternativos de gestão de recursos. 36

4.1 Diferenças entre os modelos alternativos de gestão de recursos e o modelo proposto. 55

Capítulo 1

Introdução

1.1 Contexto geral

Atualmente as necessidades dos usuários de computador são as mais variadas. É comum um usuário querer ouvir um arquivo de música, redigir um texto, fazer uma apresentação ou editar uma imagem. Para que ele possa realizar essas tarefas é necessário que o computador utilizado possua os programas que as executam. Um programa é um conjunto finito de instruções com o propósito de resolver um problema específico.

Para fazer uma melhor utilização do computador, os usuários costumam executar mais de um programa ao mesmo tempo. Por exemplo, eles mantêm o programa leitor de e-mails aberto, enquanto ouvem músicas e digitam textos. Para que a execução de um processo não interfira na execução de outro, o sistema operacional controla a ordem e o tempo de execução de todos os processos, bem como o acesso aos recursos do sistema computacional.

Com o aumento da capacidade de processamento dos computadores, os usuários habituaram-se a abrir mais programas ao mesmo tempo. Os sistemas operacionais devem estar aptos para tratarem essa grande quantidade de processos disputando os recursos computacionais para que todos possam ser executados de forma justa e adequada.

Ao se tratar de sistemas multi-usuários a situação é ainda mais crítica, pois vários usuários podem estar executando vários processos, sendo que cada processo vai concorrer pelo acesso aos recursos. Como um número muito elevado de processos podem estar em execução em um determinado instante, sistema operacionais multi-usuários devem se

preocupar mais ainda com a gestão dos recursos com a finalidade de tornar a execução de todos os processos a mais apropriada.

1.2 Motivação

Os computadores modernos consistem em processadores, memórias, temporizadores, discos, mouses, interfaces de rede, impressoras e uma ampla variedade de outros dispositivos. A função do sistema operacional, enquanto gerenciador de recursos, é oferecer uma alocação ordenada e controlada dos processadores, memórias e dos dispositivos de entrada e saída entre os vários programas que competem por eles. Quando um computador tem múltiplos usuários, a necessidade de gerenciar e proteger os recursos computacionais é maior ainda [TW97].

O objetivo principal do gerenciamento de recursos é prover oportunidade e garantia de acesso aos recursos, ao mesmo tempo que os protege. Os recursos em geral não são completamente independentes uns dos outros, assim, existem situações onde o acesso a certos recursos compromete os novos acessos a ele mesmo ou a outros [GR02].

Uma situação onde a interdependência dos recursos pode ser notada é no tratamento dos pacotes de rede. Quando a interface de rede recebe um pacote, ela faz uma requisição para o sistema operacional executar as ações necessárias para transferência do pacote para o seu destino. Porém, quando o processador está saturado, essa requisição pode demorar para ser atendida, e o pacote pode ser descartado. Isso ocorre principalmente em interfaces de rede muito rápidas, como as gigabit ethernets, em computadores com processadores lentos.

Nos sistemas operacionais de mercado existem algumas ferramentas para monitorar e controlar os recursos do sistema, mas funcionam de uma forma primitiva [MR95]. Esses sistemas operacionais geralmente fornecem somente um mecanismo de prioridades para ordenar o acesso aos recursos. E ainda assim, por muitas vezes, os ajustes de utilização dos recursos podem ser aplicados somente estaticamente, ou seja, somente na próxima execução do processo, ou no próximo login do usuário é que esses ajustes terão efeito.

Este trabalho visa definir um modelo dinâmico de controle de recursos. Por dinâmico, entende-se um modelo de controle de recursos em que não é necessário reiniciar o processo ou a sessão do usuário para que os ajustes de acesso aos recursos sejam efetuados. O

sistema aplicará as alterações assim que seja possível, sem que seja necessário realizar nenhuma outra ação. Também consiste de um modelo mais flexível de controle onde o administrador pode fixar valores de limite máximo e mínimo de utilização dos recursos pelos processos.

Com o controle dinâmico dos recursos muitas novas características podem ser incorporadas ao sistema operacional. Por exemplo, ao se fazer uma transmissão de dados na internet, pode-se garantir uma largura de banda de rede mínima à aplicação, e garantir também recursos mínimos para que a banda alocada não fique vazia por falta de oportunidade de se enviar dados sobre ela. Esse tipo de controle também poderia fornecer as primitivas para implementar um sistema operacional com suporte à qualidade de serviço (QoS). Construindo-se um escalonador de QoS que seja capaz de analisar as requisições de alocação de recursos das aplicações, o controle dinâmico poderia ser utilizado para garantir os recursos mínimos para a execução adequada dos processos.

Da mesma forma, interagindo com programas que detectam anomalias no sistema computacional, como sistemas de detecção de intrusão ou falhas, o controle dinâmico de recursos poderia ser utilizado para restringir a execução de processos comprometidos. Assim, se for detectado algum tipo de ataque a processos, podem ser lançadas medidas de contenção, não permitindo que os processos afetados sejam executados ou que sejam executados muito lentamente, evitando que ataques efetuados contra o sistema computacional tenham sucesso, inclusive bloqueando ataques de *Denial of Service* [dAMF03]. Isso garantiria ao sistema operacional mais confiabilidade e segurança.

Também pode-se privilegiar outros processos para que eles possam ter uma execução adequada. Essas são características desejadas em um sistema operacional de rede, pois muitos usuários competem por recursos compartilhados, e, dessa forma, o sistema operacional, por muitas vezes, deve ceder mais recursos a um e negar a outros, para que todos possam ser justamente executados. Podendo privilegiar determinados processos, o administrador do sistema tem uma maior flexibilidade para definir quais são as aplicações que devem ser mais prontamente executadas, focando mais diretamente no objetivo do sistema.

1.2.1 Objetivo geral

O objetivo desse trabalho é apresentar um novo modelo para controle dos recursos do sistema computacional. Esses recursos constituem-se do processador, memória, espaço em disco, banda de disco, banda de rede e recursos internos do sistema operacional. Nesse modelo os processos podem ter aumentadas ou diminuidas suas chances de acesso aos recursos a qualquer momento sem que haja a necessidade de reiniciar o processo ou a sessão. Também é possível fixar um limite máximo ou mínimo de utilização de um ou mais recursos por um processo, por um grupo de processos, por um usuário ou por um grupo de usuários.

1.2.2 Objetivos específicos

- Definir um modelo de controle que seja capaz de ser ajustado dinamicamente de acordo com as especificações do administrador para limitar ou privilegiar o acesso aos principais recursos de um sistema computacional[1];
- Especificar uma API oferecendo a possibilidade de controlar dinamicamente:

    1. o tempo de utilização dos processadores, diminuindo ou aumentando o percentual de utilização do processador, dentro de um determinado período ou durante toda a execução do processo;
    2. a memória, definindo a quantidade de páginas residentes em memória física, o número de faltas de página que ocorre e a quantidade de páginas em memória virtual;
    3. espaço em disco, definindo cotas de tamanho de arquivos e quantidade de arquivos abertos;
    4. banda de disco, especificando a quantidade de informação que pode ser transferida para o disco por segundo;
    5. banda de rede, especificando a quantidade de *bits* que podem ser enviados pela rede por segundo;

[1] Note-se que foram escolhidos os recursos mais relevantes para compor o conjunto de elementos que esse modelo aborda com base em conhecimento tácito.

6. recursos internos (descritores, sockets, semáforos e memória compartilhada), definindo a quantidade de cada recurso interno que podem ser mantidos abertos por um ou mais processos.

- Como protótipo, implementar uma API para controlar dinamicamente o processador no sistema operacional Linux.

O produto final desse trabalho consiste de alterações no núcleo do Linux para que ele possa atuar como um gerente de recursos que libere e negue acesso aos recursos e que ajuste os recursos previamente alocados, privilegiando o acesso a partir de um ou mais processos ou restringindo-os.

1.3 Estrutura do texto

Esse texto está estruturado da seguinte forma: neste primeiro capítulo foi visto a necessidade de um sistema operacional que possa controlar dinamicamente os recursos computacionais, que é a proposta desse trabalho. No segundo capítulo será visto quais são os principais recursos do computador e os métodos atuais de controle de recursos. Seguindo, o capítulo três apresentará os problemas encontrados na gestão dos recursos nos sistema operacionais atuais e será descrito alguns modelos alternativos de gerência de recursos. O capítulo quatro desmonstrará o modelo de controle de recursos proposto neste trabalho e o capítulo quinto a implementação desse modelo para o processador, com avaliação de seu funcionamento. Finalmente serão apresentadas as considerações finais e trabalhos futuros.

Capítulo 2

Gestão de recursos

2.1 Introdução

Neste capítulo serão apresentados os principais recursos que compõe um sistema computacional e como um sistema operacional convencional coordena e realiza a alocação desses recursos de forma que os processos possam em algum momento utilizá-los.

Em termos gerais, um sistema computacional é composto de diversos componentes físicos e lógicos que são utilizados por programas para serem executados. Esses componentes são conhecidos como recursos computacionais e incluem o processador, a memória, os discos rígidos, os componentes de rede, as impressoras e outros dispositivos de entrada e saída. Além destes, são também considerados recursos as estruturas lógicas providas pelo sistema operacional às aplicações, como descritores de arquivos, sockets, semáforos, *threads* e outras coisas mais.

Nos sistemas modernos é normal que existam vários processos em execução ao mesmo tempo. Sendo os recursos limitados, os processos devem acessar um determinado recurso a fim de realizar uma tarefa e, em seguida, liberá-lo para que outros processos possam também completar sua execução. Para evitar excessos na utilização dos recursos por parte de algum processo, o sistema operacional age como gerente dos recursos, prevenindo que haja abuso na utilização dos mesmos.

A seguir serão descritos os principais recursos de um sistema computacional e como o núcleo do sistema operacional gerencia o acesso a cada um deles.

2.2 Recursos de um sistema de computação

Recurso é definido como sendo um meio utilizado para atingir um fim. Em computação são os componentes físicos ou lógicos utilizados por processos para realizar alguma tarefa específica. Os recursos físicos mais comuns são o processador, a memória, discos e rede. Entre os recursos lógicos os mais comuns são os descritores de arquivos, semáforos, sockets, processos, *threads* e memória compartilhada. Nos próximos tópicos, estarão relacionados os principais recursos físicos e lógicos, bem como, será mostrado os métodos mais comuns utilizados pelos sistemas operacionais para fazer o seu controle.

2.2.1 Gestão do processador

O processador é o componente crucial de um sistema computacional. É nele que ocorre a atividade fim de qualquer sistema informático, é onde realmente ocorre o processamento dos dados.

Na realidade, são duas as funções básicas do processador [Mon02]:

- função de processamento;
- função de controle.

As tarefas mais comuns que são executadas pelo processador em sua função de processamento são as operações aritméticas, operações lógicas, movimentação de dados, desvios e operações de entrada e saída.

Na função de controle, as principais atividades desenvolvidas pelo processador são a busca das instruções a serem executadas, interpretação das ações a serem tomadas e o controle dos componentes internos, como a unidade lógica e aritmética, e dos componentes externos como memória e dispositivos de entrada e saída.

Devido a seu papel fundamental na execução das instruções, o processador é um dos recursos mais requisitados no sistema computacional. Cada processo necessita obter acesso ao processador para executar as suas instruções, seja para fazer algum cálculo, imprimir um documento ou simplesmente para encerrar sua execução. Mesmo em sistemas computacionais de escritório são vários os processos em execução simultânea, sendo a disputa pelo uso do processador intensa. Em sistemas operacionais de rede, que geralmente fornececem uma grande quantidade de serviços e/ou uma grande quantidade de acessos, o

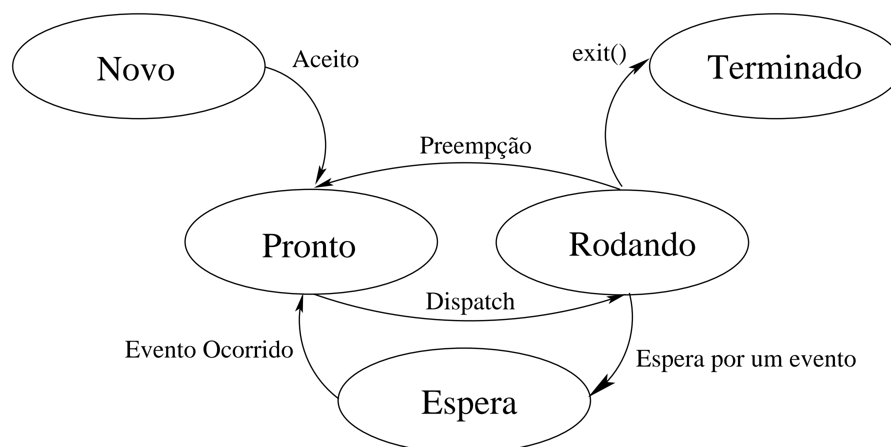

Figura 2.1: Diagrama de estado dos processos.

número de processos sendo executados ao mesmo tempo pode ser ainda superior, tornando a competição pelo uso do processador ainda mais intensa.

O responsável por controlar quem deverá obter o acesso a esse recurso é o sistema operacional. É ele que define quanto tempo cada processo tem para executar suas instruções no processador e quando deverá voltar a recebê-lo após terminado seu período de execução.

Sendo o sistema monoprocessado, isto é, possuindo somente um processador, não é possível executar mais que um processo por vez. Assim é necessário que o sistema operacional implemente um mecanismo que permita uma pseudo execução paralela dos processos. São os chamados sistemas operacionais multitarefas. Os processos são armazenados em memória durante a carga do programa e, então, o sistema operacional alterna os processos que estão sendo executados de tempo em tempo, executando parcialmente cada um em sua vez. A parte responsável por essa tarefa dentro de um sistema operacional é o escalonador. Com isso tem-se vários programas sendo executados em tempos diferentes, porém com a sensação de estarem sendo executados simultaneamente, devido à rápida alternância entre eles.

A seguir serão vistos os principais algoritmos utilizados pelos sistemas operacionais para implementar o escalonador de processos.

Escalonadores Round Robin

O Round Robin é um dos algoritmos de escalonamento mais antigos e mais simples e consiste na atribuição de uma fatia de tempo para execução do processo, denominado

quantum, que na verdade é um valor que corresponde a quantos *ticks*[1] de relógio aquele processo tem disponível para usar o processador. Cada vez que ocorre um *tick* do relógio, o processo que está sendo executado tem seu *quantum* decrementado. Quando esse valor for igual a zero o processo é retirado de execução, indo para uma fila de processos que estão prontos para serem executados, e estão esperando a sua vez de obterem o processador, enquanto outro processo é colocado em seu lugar. Isso é chamado de *troca de contexto*.

Se um processo que está de posse do processador faz uma requisição de entrada ou saída, sendo que o tempo de acesso aos dispositivos de entrada e saída é muito alto, ou lança algum outro tipo de evento a qual ele deve esperar para ser atendido, como um acesso a uma região crítica ou um alarme, esse processo perde o processador e é colocado em uma fila de processos bloqueados e que estão esperando pelo evento que satisfará a sua requisição. Quando esse evento ocorrer ele será colocado novamente na fila de processos prontos para execução, voltando a concorrer ao uso do processador.

Nesse tipo de escalonamento todos os processos recebem sempre o mesmo *quantum*, sendo tratados como iguais. Não há nenhuma prioridade de execução de um processo sobre outro. Assim sendo, um processo de pouca importância, um desfragmentador de disco por exemplo, obtém o mesmo tempo de execução que um processo relevante, como um servidor web, ignorando toda e qualquer hierarquia entre os processos.

De fato, o escalonamento por esse algoritmo consiste de uma fila de processos prontos para serem executados. O processo escolhido para utilizar a CPU é sempre o primeiro da fila, e ao terminar o seu *quantum*, ele é colocado ao final da fila, e o processo que agora ocupa a primeira posição na fila é então pego para ser executado, como pode ser visto na figura 2.1.

A principal característica desse algoritmo é a utilização do *quantum*, que não deve ser muito pequeno, pois geraria uma sobrecarga muito grande no uso do processador por parte do escalonador devido as freqüentes trocas de contexto, e também não deve ser muito grande, pois a demora em realizar a troca de contexto pode tornar o tempo de resposta do sistema muito elevado.

[1] Um tick é definido como sendo o intervalo de tempo entre duas interrupções consecutivas do relógio do hardware. Esse intervalo de tempo também é conhecido como ciclo de tempo.

Escalonamento por prioridade

Conforme descrito, o algoritmo Round Robin trata todos os processos como se todos tivessem a mesma importância para o sistema. Isso, no entanto, não é sempre verdade para os usuários, nem mesmo para o sistema computacional ou para a organização, pois alguns processos por muitas vezes são muito mais relevantes do que outros.

Houve então a necessidade de criar prioridades para a execução de processos, de forma que processos de maior importância pudessem obter o processador antes de outros, sendo, então, executados mais rapidamente, de acordo como foi definida a sua relevância.

A fim de evitar que processos de alta prioridade sejam executados indefinidamente, não permitindo que nenhum outro possa também ser executado, pode-se diminuir a prioridade dos processos de mais alta prioridade de acordo com o tempo que eles obtêm o processador. Assim, mesmo os processos com baixa prioridade continuarão sendo executados, porém menos freqüentemente que os processos de maior prioridade. Isso é chamado de envelhecimento de processos. Obviamente, o processo tem sua prioridade restaurada assim que uma determinada condição seja atingida.

Outro mecanismo para implementar prioridades, e ainda assim permitir que todos os processos sejam executados, é atribuir valores de *quantum* diferentes de acordo com a prioridade dos processos. Processos com maior prioridade obtêm um *quantum* maior, utilizando a CPU por mais tempo.

As prioridades dos processos não precisam ser estáticas, podendo, então, ser alteradas de acordo com as necessidades do usuário, do sistema computacional ou da própria instituição que o mantém. Também é possível que o sistema altere automaticamente a prioridade de seus processos a fim de atingir uma determinada meta. Em sistema Unix, por exemplo, processos orientados a entrada e saída são priorizados em relação aos processos orientados a CPU.

Escalonamento *job* mais curto primeiro — SJF

Esse é um mecanismo especialmente utilizado em sistemas em que processos são executados em lote, onde previamente se conhece o tempo de execução do processo. A idéia de sistemas em lote é que todos os processos que precisam ser executados sejam colocados juntos em uma unidade de entrada e, assim, o primeiro processo é executado até a sua

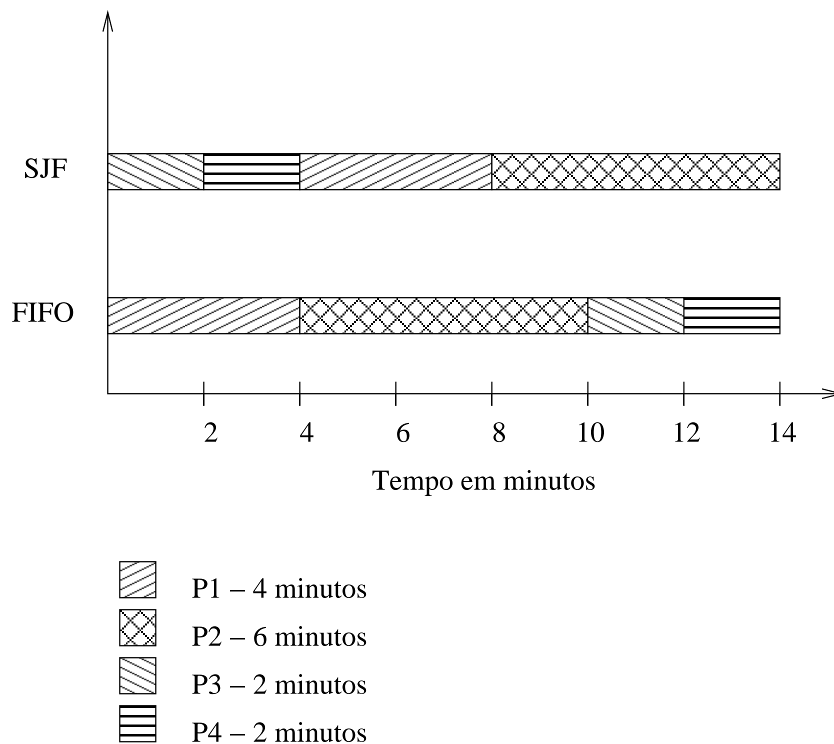

Figura 2.2: Ordem de execução de processos pelo algoritmo SJF.

finalização, seguido de outro e assim por diante. Como esses processos são constantemente executados pode-se predizer o tempo de execução de cada um deles.

Nesse algoritmo, o processo que deve primeiramente ser escolhido para execução é o processo mais curto, isso porque sendo executado primeiro o tempo médio de resposta total do sistema diminui. Um exemplo desse algoritmo é dado a seguir, conforme a figura 2.2: sejam os processos P1, P2, P3 e P4 com tempo de execução em minutos de 4, 6, 2, 2 respectivamente, o tempo de resposta do sistema se os processos forem executados nessa ordem, é de 4 minutos para P1, 10 para P2, 12 para P3 e 14 para P4, sendo que o tempo médio é então de $(4 + 10 + 12 + 14)/4$, ou seja de 10 minutos.

Se for utilizado o escalonador SJF, os processos P1, P2, P3 e P4 serão executados na seguinte ordem: P3, P4, P1, P2, sendo que o tempo de resposta para P3 é de 2 minutos, P4, 4, P1, 8 e P2, 14. O tempo de resposta médio então é $(2 + 4 + 8 + 14)/4$, ou seja de 7 minutos.

Como SJF produz sempre o menor tempo médio de resposta, seria bom se ele pudesse também ser utilizado em sistemas interativos, no entanto, para isso precisa-se prever o tempo de execução de cada processo e descobrir qual entre eles é o mais curto, o que é difícil de se conseguir, no entanto podendo ser feito no nível de *quantum*.

Escalonamento de tempo-real

Sistemas operacionais de tempo-real são aqueles onde os resultados dos processos para serem considerados corretos, devem, além de estarem logicamente adequados, ser produzidos em um prazo previamente estipulado.

Esses sistemas são utilizados em áreas específicas, onde o atraso na resposta dos processos do sistema pode provocar um impacto tão ruim quanto não os executar. São os casos dos sistemas de controle de segurança de veículos automotivos modernos, de espaçonaves, aviões, sistemas de monitoramento em unidades de tratamento intensivo, etc.

Nos sistemas de tempo-real críticos (*hard real-time*), o não cumprimento do requisito temporal pode ser catastrófico tanto no sentido econômico quanto em vidas humanas [OCT02]. Um exemplo de utilização desse método seria os sistemas operacionais das baterias anti mísseis que necessitam reagir no tempo exato para cumprir o seu papel.

Caso o sistema aceite um atraso tolerável na execução dos processos, o sistema é conhecido como sistema de tempo-real flexível (*soft real-time*). Pode ser utilizado, por exemplo, na execução de aplicações multimídia, onde o não cumprimento das metas estipuladas não causaria maiores danos, somente a diminuição da qualidade de apresentação do programa.

Os sistemas de tempo-real podem ser implementados de forma que as decisões de escalonamento são conchecidas antes mesmo que os processos estejam em execução. Por exemplo, o algoritmo de escalonamento sabe que deve executar o processo P1 antes de todos em qualquer situação, caso P1 não esteja presente ele escalonará os outros processos de acordo com o mecanismo implementado. Essa técnica é conhecida como escalonamento estático, e geralmente é usada quando se possui um grupo de processos pequeno e bem definido.

O escalonamento dinâmico consiste em tomar as decisões de escalonamento durante a execução dos processos de acordo com algum parâmetro, como prioridade entre os processos, o prazo final ou o tempo que pode ser dispensado para execução dos processos.

Para o Linux existem implementações tanto de sistemas de tempo-real flexíveis, como os [CI01, Jac02], quanto os críticos, como os [DNFW02, Yod99].

Processos e threads

Em computação, um processo é uma instância em execução de um programa, incluindo todas as suas variáveis e estados [FM02]. Em geral o processo é visto como *container* de recursos como memória — código e dados —, descritores de arquivos, semáforos, etc.

Na maioria dos sistemas operacionais de mercado a unidade básica de utilização do processador não é processo, e sim *thread*. As *threads* pertencem aos processos e compartilham com ele código, dados e outros recursos que pertencem ao processo, no entanto, possuem seu próprio contador de programa, conjunto de registradores e pilha, e podem ser executadas separadamente. As *threads* são geralmente usadas em situações onde tarefas similares precisam ser realizadas ou quando tarefas necessitam ser executadas simultaneamente dentro do processo. Por exemplo, um servidor de e-mail que possa receber conexão de vários clientes. Essas conexões podem ser gerenciadas por *threads*, uma para cada conexão, para que o processo do servidor de e-mail possa atender a várias requisições ao mesmo tempo.

A utilização de *threads* melhora a interatividade de uma aplicação, já que uma *thread* do processo pode ser, por exemplo, bloqueada, enquanto o processo continua as suas demais operações normalmente. Também pela economia de tempo na criação dos processos e na troca de contexto, visto que a *thread* utilizará os recursos já criados para o seu processo. Em sistemas multiprocessados, cada *thread* pode estar rodando em paralelo, diminuindo o tempo de execução total.

As *threads* são recursos lógicos do sistema operacional e podem ser implementadas conforme os modelos abaixo:

Modelo Muitos-para-um : o gerenciamento das *threads* nesse modelo é feito no espaço do usuário. Também são conhecidas como *threads* de usuário. É impossível executá-las em paralelo em sistemas multiprocessados. Quando uma *thread* de usuário é bloqueada todas as demais *threads* do mesmo processo são bloqueadas também;

Modelo Um-para-um : nesse modelo cada *thread* do usuário é mapeada em uma *thread* do núcleo. Cada *thread* é executada separadamente, podendo se beneficiar do paralelismo em sistemas multiprocessados. O bloqueio de uma *thread* não implica no bloqueio do processo como um todo. O Windows NT, o OS/2 e o Linux implementam esse modelo;

Modelo Muito-para-muitos : nesse modelo as *threads* de um processo do usuário são combinadas em um menor ou igual número de *threads* no núcleo. Os escalonamento é feito tanto no nível do usuário quanto no nível do núcleo. Esse modelo é mais flexível, no entanto muito mais complexo, exigindo que o programador defina quantas *threads* o núcleo terá para o processo e como será feita a sua execução. Os sistemas operacionais IRIX, o Solaris e o Digital Unix utilizam esse modelo.

2.2.2 Gerência de memória

Memória é um componente computacional que permite armazenar dados e códigos de programas temporariamente a fim de permitir a execução dos processos na CPU. É também conhecida como RAM, ou memória de acesso aleatório. Na arquitetura atual dos computadores, um processo para ser executado necessita estar em memória, pois é nela que o processador busca as instruções para execução. Cada processo obtém uma quantidade finita de memória de acordo com sua própria necessidade. Assim sendo, quanto mais memória o computador possui, mais processos podem ser armazenados nela simultaneamente, tornando um sistema multiprogramado mais eficiente. Os dados e os códigos são armazenados na memória de forma volátil, isto é, eles são perdidos assim que a memória pára de receber energia.

Sendo a memória um componente de tamanho finito (256MBs, 512MBs, etc.), não é possível armazenar nela infinitos processos, nem tampouco processos muito grandes que não possam se ajustar à memória. Caso a memória não seja suficientemente grande para alocar todos os processos, o sistema pode utilizar o disco rígido para armazenar o código e os dados dos processos enquanto não estão sendo executados. A utilização do espaço em disco mais a memória física como área de alocação é conhecida como memória virtual.

Como o acesso ao disco é muito mais lento do que o acesso à memória, computadores com pouca quantidade de memória tem a sua eficiência comprometida devido ao grande uso de disco. Assim, processos que se mantêm o tempo todo em memória física são mais rapidamente executados em comparação aos que utilizam o disco. O espaço em disco utilizado para armazenar informações da memória física é chamado de área de *swap*.

O método mais utilizado para implementação da memória virtual é o da paginação, que consiste no mapeamento dos endereços de memória internos do processo, os endereços

virtuais, em outros endereços, os reais ou físicos. Ou seja, os endereços de memória que os processos referenciam são posições de memória relativas a endereços físicos. O componente computacional responsável pelo gerenciamento da memória virtual é a MMU (Memory Management Unit), Unidade de Gerenciamento de Memória.

Os endereços gerados pelos processos, os endereços virtuais, formam o espaço de endereço virtual que é dividido em páginas. As páginas são unidades de memória onde os processos são armazenados. A memória física é dividida em quadros [TW97] que possuem o mesmo tamanho das páginas e podem armazenar o conteúdo de uma página.

Quando um programa é executado, seu código e seus dados são colocados nas páginas de memória, que podem estar tanto em *swap* quanto na memória. A medida em que é requerido o código ou os dados dos processos que estão em páginas armazenadas em disco, ocorre uma interrupção chamada de falta de página que notificará o sistema operacional que uma página necessária para a continuidade de execução do processo não está em memória. Caso não haja mais quadros vazios, ou seja, toda a memória física está alocada, um algoritmo escolhe uma das páginas em memória para ser guardada em disco e o quadro relativo a sua posição é então utilizado. Com isso, os processos podem até ocupar um espaço maior que o tamanho da memória física e eles ainda assim serão executados. A figura 2.3 apresenta a relação entre os endereços internos aos processos com os endereços reais do computador.

Quando ocorre uma falta de página, o sistema deve escolher uma página que irá ser removida da memória para dar lugar à página que necessita ser carregada. Existem vários algoritmos de escolha da página que deve ser retirada de memória. A seguir são apresentados os algoritmos mais utilizados para esse fim.

Algoritmo ótimo

O algoritmo ótimo seria aquele que descarta a página que levará mais tempo para ser referenciada pelos processos entre todas as outras que estão na memória física. Este algoritmo seria o ótimo, pois não gastaria tempo computacional retirando uma página que vai ser utilizada logo em seguida. No entanto, este algoritmo é impossível de se implementar, pois os sistema operacional não sabe quais páginas serão referenciadas nas próximas instruções.

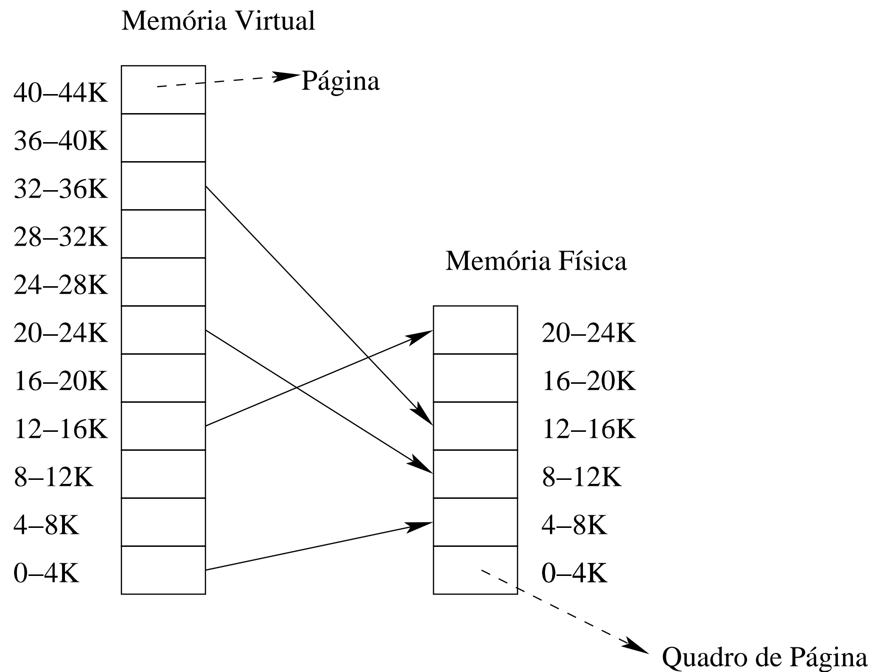

Figura 2.3: Relação entre endereços virtuais e endereços físicos.

First In, First Out — FIFO

Esse algoritmo consiste em descartar as páginas mais antigas que estão em memória. Para isso, o sistema operacional guarda informações da ordem de entrada da página em memória através de uma lista, na qual o seu início é a referência para a página mais antiga. Quando ocorre uma falta de página, a página referenciada pelo início da lista é retirada da memória e a página que precisa ser carregada é inserida no final da lista.

Esse algoritmo, apesar de funcional, não é eficiente, pois estar em memória a mais tempo não é sinônimo de não ser utilizada. Assim sendo, se a página que se encontra no início da lista for muito utilizada, logo deverá ser carregada novamente em memória, disperdiçando tempo de execução.

Segunda chance

Uma variação da implementação do FIFO é verificar se a página referenciada pelo início da fila não foi utilitizada recentemente, então ela pode ser imediatamente substituída. No entanto, se ela foi utilizada, ela é colocada no final da fila. O sistema, então, verificará se a página referenciada pelo nó que agora se encontra no início da fila foi também recentemente utilizada e a substituirá ou a colocará no final da fila de acordo com o resultado do teste,

e assim por diante.

Esse algoritmo possui uma leve vantagem sobre o FIFO em relação a troca de páginas que estão sendo utilizadas, porém, se todas as páginas estiverem sendo utilizadas, o algoritmo se tornará um FIFO puro.

Não recentemente utilizada

O hardware para controle de memória virtual implementa um mecanismo para saber o estado de cada página, ou seja, ele guarda informações sobre as páginas referenciadas pelos processos. Para isso, cada página possui um bit R que quando setado indica que a página foi utilizada. Quando ocorre um *tick* do relógio o bit R é zerado. Isso permite que o sistema saiba se a página foi recentemente utilizada. O hardware também guarda a informação de modificação da página em um bit chamado M. Este bit é setado caso a página tenha sido modificada, para que no momento de sua substituição o sistema saiba se ela necessita ser gravada novamente em disco. O bit M não é zerado quando ocorre o *tick* do relógio, pois senão uma página modificada poderia ser descartada.

Esse algoritmo divide as páginas que estão em memória em classes de acordo com as informações do estado de cada uma delas:

- classe 0: não-utilizada, não-modificada;
- classe 1: não-utilizada, modificada;
- classe 2: utilizada, não-modificada;
- classe 3: utilizada, modificada.

Assim, o algoritmo escolhe uma página da classe de numeração mais baixa para dar lugar à nova página que está sendo carregada. Caso nenhuma página se encontre nessa classe, uma página da classe superior é escolhida.

Com isso, o algoritmo consegue escolher páginas que não foram recentemente utilizadas, antes de descartar uma que está sendo utilizada constantemente. Não é um algoritmo ótimo, mas é freqüentemente adequado.

Menos recentemente utilizada — LRU

Outro método que tem uma boa aproximação do algoritmo ótimo para troca de páginas é retirar de memória a página que foi menos recentemente utilizada, ou seja, a página que permaneceu mais tempo sem ser referenciada por nenhuma instrução.

Para que seja possível a implementação desse algoritmo é necessário que o hardware a cada instrução incremente um contador específico para a página em memória que foi utilizada naquela instrução.

Quando ocorrer uma falta de página, o sistema operacional escolherá para substituição a página com o menor valor em seu contador. O mesmo mecanismo pode ser implementado utilizando-se matrizes. Um método semelhante pode ser implementado via software utilizando os valores de R, que é o bit que armazena a informação de utilização da página.

Alguns sistemas operacionais de mercado permitem setar o número máximo de páginas em memória que um processo pode ter, bem como setar o máximo de espaço de endereços de um processo.

Como manter o processo em memória física incrementa sua velocidade, visto que não faz uso de disco, é importante ter um mecanismo que privilegie o uso da memória por processos específicos, conforme sua necessidade.

2.2.3 Gestão de disco

O disco é um dispositivo para armazenamento de dados de forma não-volátil. Um disco é na realidade um conjunto de superfícies circulares magnéticas que são rotacionadas juntas. Em ambos os lados de cada superfície existe uma cabeça de leitura e gravação responsável por ler e escrever dados. Essas cabeças se movimentam juntas, isto é, se uma cabeça estiver sobre determinada posição na superfície, todas outras também estarão na mesma posição em suas respectivas superfícies. Cada superfície é dividida em trilhas, que são áreas circulares concêntricas. Elas são divididas em pedaços menores denominados setores. São neles que as informações são armazenadas. Os setores também são as unidades de transferência de dados entre o disco e o sistema computacional. O conjunto de trilhas de mesma posição nas diferentes superfícies é chamado de cilindro.

Cada disco possui um tamanho que é a quantidade máxima de bytes que podem

ser armazenados nele, geralmente medido em gigabytes (GBs). A banda de um disco é a quantidade máxima de dados que podem ser transferidos entre o disco e o sistema computacional, sendo medido em megabytes por segundo (MBs/s).

Em sistemas operacionais Unix é possível limitar o espaço em disco utilizado por um usuário ou grupo de usuários, bem como limitar o número de *inodes*[2] utilizado por eles [Kar01]. Para tanto deve-se definir o limite desejado e habilitar o sistema de cotas através do conjunto de utilitários *quota*.

Embora exista o sistema de cotas para limitar o uso do espaço em disco na maioria dos sistemas operacionais de mercado, não há nenhuma maneira de controlar a utilização da banda de disco [CI01]. Isso se deve ao fato de que o algoritmo de priorização de acesso ao disco é feito com base na posição em que se deseja acessar os dados no disco. A seguir serão descritos os principais algoritmos para agendamento das requisições de gravação e leitura em disco.

First-Come, First-Served — FCFS

Nesse algoritmo o processo que vai primeiramente ter a sua requisição de escrita ou leitura em disco satisfeita é aquele que primeiro requisitou uma ação no disco. Esse algoritmo não tem como objetivo otimizar o tempo de busca ou mesmo do deslocamento da cabeça do disco.

Shortest Seek First — SSF

Nessa técnica os dados que estão próximos da cabeça de leitura do disco são lidos ou gravados antes, e o processo a quem eles pertencem é privilegiado. Com esse algoritmo é possível diminuir muito o deslocamento da cabeça do disco. No entanto, quando há uma sobrecarga de requisições a cabeça tenderá a ficar sobre uma determinada área do disco, pois tentará atender sempre as requisições que estão mais próximas, gerando um conflito entre o deslocamento da cabeça do disco e o tempo de resposta mínimo para as requisições e a justiça no acesso ao disco entre os processos.

[2] Inode é uma estrutura de dados que armazena informações sobre os arquivos em um sistema de arquivo Unix. Existe um inode para cada arquivo. Um arquivo é identificado de forma única pelo sistema de arquivos que ele reside e pelo número de seu inode nesse sistema [Fou01].

Algoritmo do elevador

Como o nome já diz, esse algoritmo possui a mesma política de decisão de deslocamento da cabeça de gravação que os elevadores convencionais. Basicamente, a cabeça do disco permanece na mesma direção que ela está se movimentando até que todas as requisições que se encontram à sua frente naquela direção sejam atendidas ou que o braço atinja o limite físico do hardware. Quando isso acontece a direção de movimentação é invertida e as requisições vão sendo atendidas até que não haja mais nenhuma requisição pendente naquela direção. Esse algoritmo além de minimizar o deslocamento da cabeça do disco, não implica em diminuição do tempo de resposta mesmo quando há uma sobrecarga de requisições.

No Linux, por exemplo, o algoritmo utilizado para o agendamento das requisições de ações em disco é o algoritmo do Elevador, otimizando o tempo de busca e aumentando o desempenho de entrada e saída. No entanto, como na maioria dos sistemas operacionais modernos, não há como priorizar o atendimento das requisições dos processos de maior importância para a organização ou para o usuário.

2.2.4 Gestão de rede

A interface de rede é um dispositivo que conecta o computador com outros computadores ou periféricos usando algum tipo de meio de comunicação, como cabo coaxial, par-trançado, rádio, fibra óptica, etc. Existem vários padrões de comunicação entre interfaces de rede, sendo a mais comum a Ethernet. Com um dispositivo de rede um computador é capaz de enviar e receber dados encapsulados em forma de pacotes.

Os processos que desejam utilizar a rede fazem requisições de envio de pacotes ao núcleo do sistema operacional. Se houver várias requisições de envio no mesmo período de tempo, o núcleo deve decidir qual o pacote deverá ser enviado primeiro, quais vão esperar e quais serão descartados. O mesmo pode ocorrer com os pacotes que são recebidos pela interface de rede. A tarefa de seleção de pacotes é realizada por um escalonador de pacotes que pode ser implementado de várias maneiras.

O escalonador mais utilizado nos sistemas operacionais de mercado é o *First In First Out*, ou seja, será atendida a requisição que aconteceu primeiro. Nesse algoritmo não existe uma política de atender antes processos com maior prioridade, nem existe controle

da banda utilizada por um processo ou usuário. O seu objetivo é simplesmente transmitir os pacotes pela rede da forma mais simples.

Entende-se por banda a quantidade de dados que podem trafegar entre a interface local de rede e outro ponto na rede. A banda de rede é geralmente medida em megabits por segundo (Mbs/s).

Alguns sistemas operacionais modernos, como o Linux e o Windows XP, já implementam outros algoritmos de escalonamento de pacotes que possuem um controle do recurso mais sofisticado, permitindo controlar a banda de rede.

No Windows, pode-se habilitar o *QoS Packet Scheduler* que implementa o algoritmo *Deficit Round Robin (DRR)*, que aloca vários canais de fluxos de dados e associa os programas a estes canais. É possível fazer reserva de banda de rede utilizando a API de QoS do Windows.

No Linux vários algoritmos são implementados para QoS. O algoritmo CBQ (*Class-Based Queueing*) [FJ95] classifica os pacotes que estão esperando em uma hierarquia estilo árvore; as folhas dessa árvore são escalonadas por diferentes algoritmos. A grande vantagem do CBQ, é permitir o controle não só do tráfego de entrada, bem como o trafego de saída. Já o algoritmo *Traffic Shapper* é útil para reduzir a taxa do fluxo de saída de dados pela rede. Ele faz isso criando dispositivos especiais virtuais e os associando aos dipositivos de rede físicos.

Para habilitar esses algoritmos é necessário configurá-los através de ferramentas específicas. No Linux, o administrador pode controlar esses algoritmos com os utilitários `iproute2+tc`.

Através dessas ferramentas é possível determinar a banda disponível para uma determinada porta[3], que está sempre associada a um processo. Assim, se se quer aumentar ou diminuir a banda destinada a um processo, cria-se uma regra para aquela porta.

Esses algoritmos são implementados tanto para fornecer qualidade de serviço — QoS — em rede, como também para uma melhor configuração do sistema quando trabalhando como roteador.

Esses algoritmos associados com outros protocolos podem ser uma grande ferramenta para controle dos recursos. Inclusive, eles podem ser utilizados juntos com RSVP [ZDE+93] que é um protocolo de sinalização de reserva de recursos, para garantir que

[3]Porta é o nome dado ao ponto final de uma conexão lógica de rede.

processos terão a banda de rede necessária para executar adequadamente suas tarefas.

2.2.5 Recursos internos

Descritores de arquivos

Em sistema Unix, para obter acesso aos arquivos, o processo executa a chamada de sistema `open()` que abre o arquivo e retorna um valor não negativo, conhecido como descritor de arquivo. Esse valor é utilizado para escrever ou ler dados do arquivo.

Existem vários tipos de arquivo em sistemas Unix:

- arquivos comuns;
- diretórios, que contém uma lista de arquivos comuns;
- dispositivos de blocos, como discos rígidos;
- dispositivos de caracteres, como teclados e mouses;
- pipes e sockets, que permitem processos trocar informações entre si;
- links simbólicos, que permitem um arquivo ter mais de um nome.

A maioria dos recursos computacionais são acessados através dessas abstrações de arquivos [Tro99]. Recursos como sockets, arquivos, dispositivos de som, unidades de fita, disquetes, terminais e informações do núcleo são tratados como se fossem arquivos.

O sistema Linux, por exemplo, provê uma maneira de limitar o máximo de descritores abertos no sistema inteiro ou por um processo. No diretório `/proc/sys/fs` o arquivo `file-max` mantêm o número máximo de descritores de arquivos que podem ser usados ao mesmo tempo por todos os processos do sistema. Para setar o limite máximo de arquivos abertos por um processo utiliza-se o comando `ulimit -n MAX`, onde `MAX` é o número máximo de descritores que podem ser abertos.

Sockets

Socket é uma API de comunicação com as camadas mais baixas da rede, ou seja, independente de hardware. Sockets permitem a comunicação ponto-a-ponto entre processos e é geralmente usado em implementações de serviços de rede.

O limite do número de sockets que um processo pode criar pode ser definido pelas chamada de sistema `setsockopt` em sistemas que seguem o modelo System V e o BSD.

Semáforos

Semáforos são contadores que permitem ou não acesso aos recursos compartilhados pelas *threads* ou por processos concorrentes. Suas operações básicas são de incrementar e decrementar o contador atomicamente. Quanto o semáforo possui valor positivo, é permitido o acesso ao recurso compartilhado. A contrário, quando o semáforo possui valor 0 ou negativo, o acesso ao compartilhamento é negado.

O seu limite de utilização é o número máximo de semáforos abertos no sistema ou o número de semáforos por identificador. Nenhum outro método de controle de utilização é possível.

Memória compartilhada

Memória compartilhada é uma região de memória que pode ser compartilhada por processos ou *threads* do sistema. É utilizada para comunicação entre processos e transferência de dados entre eles. Semáforos são utilizados para impedir o acesso simultâneo à memória compartilhada evitando que dados ali armazenados fiquem incorretos ou inconsistentes.

Assim como os semáforos, não existe uma forma de controlar a memória compartilhada. Um processo tem permissão para acessar ou criar um determinado número de pedaços de memória compartilhada e cada pedaço tem uma máximo de tamanho estipulado.

2.2.6 Classificação dos recursos

Quando se tratando de controle de recursos, pode-se classificar os recursos em:

- preemptíveis e não preemptíveis;
- com métricas absolutas e com métricas relativas.

Recursos preemptíveis e não preemptíveis

Os recursos preemptíveis são aqueles que podem ser alocados aos processos e retirados sem que corrompa a continuidade de sua execução. Um exemplo desse tipo de recurso é o processador, que é entregue a um processo para execução, e logo em seguida é retirado do mesmo, para ser entregue a outro. Recursos que se encontram nessa categoria são a quantidade de memória física, banda de disco e banda de rede.

Já os recursos não preemptíveis são aqueles que não podem ser simplesmente retirados do processo sem afetar a continuidade de sua execução. Se enquadram nessa categoria a quantidade de memória total (física + *swap*) destinada a um processo e o número de descritores abertos. Limitar o uso desses recursos em níveis muito baixos pode impedir que os processos sejam executados ou levá-los a erros, caso sejam negados seus pedidos de abertura de arquivos ou alocação de memória.

Recursos com métricas absolutas e com métricas relativas

Os recursos com métrica absoluta são aqueles que se pode contabilizar de maneira efetiva a sua utilização. Um exemplo é o número de descritores de arquivos abertos destinados para um determinado processo que podem ser contados em valores inteiros. Participam dessa categoria a utilização de memória física, *swap* e total, descritores de arquivos e utilização do espaço em disco.

Já os recursos com métrica relativa são aqueles que se contabilizam de forma relativa, ou seja, em função de um segundo parâmetro, geralmente o tempo. Como exemplo, tem-se a banda de disco que é calculada em função do tempo. O uso do processador, da banda de disco e banda de rede são encaixadas nessa categoria.

2.3 Conclusões

Neste capítulo foram abordados os principais recursos computacionais e como um sistema operacional pode realizar a gestão de cada um deles. No próximo capítulo serão apresentadas críticas aos mecanismos convencionais de alocação de recursos e abordagens avançadas de controle do uso dos recursos.

Capítulo 3

Mecanismos avançados de gestão de recursos

3.1 Introdução

No capítulo anterior foram vistos os principais métodos utilizados pelos sistemas operacionais modernos para gerenciar a distribuição dos recursos mais relevantes de um computador. Esses métodos são geralmente desenvolvidos de forma a satisfazer a maioria das necessidades comuns dos sistemas computacionais das organizações ou dos usuários. No entanto, esses métodos nem sempre são os mais adequados para serem utilizados em situações específicas onde se pode querer um controle mais refinado ou sofisticado do uso dos recursos.

Nesse capítulo será discutida a abordagem clássica de gestão de uso do processador em sistemas convencionais, conhecida como *escalonamento baseado em prioridades*. A seguir, abordagens alternativas permitindo um controle mais sofisticado desse recurso serão discutidas e comparadas.

3.2 Escalonamento baseado em prioridades

Um sistema operacional deve ser projetado de forma que, em situação normal, todos os processos possam obter acesso aos recursos computacionais e possam ser executados de maneira satisfatória. Os processos devem conseguir acesso aos recursos mesmo quando

muitos destes disputam o acesso àqueles, e quando não for possível atender a todas as requisições de acesso a um recurso em um determinado instante, o sistema operacional tem a responsabilidade de escolher quais processos obterão acesso ao recurso e quais deverão esperar, de maneira que todos processos tenham a garantia que serão executados, mesmo que isso comprometa o tempo de execução de um ou mais deles.

No entanto, o sistema operacional pode não fazer a melhor escolha de quem terá acesso prioritário aos recursos, pois não conhece a importância de cada um dos processos para o sistema como um todo ou para a organização que o mantém. Os sistemas operacionais de mercado implementam somente um mecanismo de prioridade determinado por um único critério para estabelecer a ordem de importância na obtenção dos recursos computacionais, ignorando que existem muitas outras variáveis que poderiam ser levadas em conta para fazer tal escolha.

Isso pode ser notado no exemplo a seguir: se vários processos desejam escrever dados no disco rígido, sendo que limitações físicas restringem a taxa de tranferência de dados, alguns desses processos devem esperar que outros processos utilizem esse recurso primeiro, para que só então eles possam guardar os seus próprios dados. Em grande parte dos sistemas operacionais a escolha dos processos que serão primeiramente atendidos é feita através de um algoritmo, conhecido como elevador, como visto na seção 2.2.3, que visa distribuir o acesso ao disco de forma homogênea entre os requisitantes ao mesmo tempo em que busca otimizar os movimentos do braço de leitura. A decisão neste caso é feita somente com base no deslocamento do braço e na uniformidade de acesso ao dispositivo, sem mesmo verificar a prioridade que o processo possui ou a sua importância para o sistema. Com isso, a escolha nem sempre é a mais adequada para a organização e nem mesmo para o sistema computacional.

A tabela 3.1 apresenta os resultados de um experimento que visa demonstrar como um sistema operacional típico distribui o processador entre vários processos requisitantes usando o escalonamento baseado em prioridades. Vários sistemas operacionais utilizam a prioridade do processo — que é um valor definido no lançamento do processo, podendo ser alterado durante a vida dele — como parâmetro de escolha de qual processo tem prioridade no uso da CPU. O primeiro teste está representado na tabela na terceira e quarta linha, e mostra a utilização do processador por vários processos (P1 a P5)

Tabela 3.1: Uso do processador por N processos com diferentes prioridades durante a inserção de um novo processo.

		P1 (-20)	P2 (-10)	P3 (0)	P4 (10)	P5 (20)	P6
Inicial		37,9%	27,6%	20,7%	10,3%	3,4%	
Inserindo P6 (-20)	Nova fatia	27,3%	19,7%	14,8%	7,4%	2,5%	27,3%
	Variação	**27,9%**	**28,6%**	**28,5%**	**28,1%**	**26,4%**	
Inserindo P6 (-10)	Nova fatia	29,3%	21,3%	16,0%	8,0%	2,7%	21,3%
	Variação	**22,9%**	**22,8%**	**22,7%**	**22,3%**	**20,0%**	
Inserindo P6 (0)	Nova fatia	31,2%	22,8%	17,0%	8,5%	2,8%	17,0%
	Variação	**17,6%**	**17,3%**	**17,8%**	**17,4%**	**17,6%**	
Inserindo P6 (10)	Nova fatia	33,0%	24,0%	20,0%	9,0%	3,0%	9,0%
	Variação	**12,9%**	**13,0%**	**13,38%**	**12,6%**	**11,7%**	
Inserindo P6 (20)	Nova fatia	36,5%	22,8%	17,0%	8,5%	11,7%	3,3%
	Variação	**3,7%**	**3,9%**	**3,8%**	**3,8%**	**2,9%**	

com diferentes prioridades de execução[1]. Então, em um determinado instante um novo processo é inserido no sistema (P6), esse com prioridade (-20), que é a mais alta em sistemas Unix. As linhas em preto mostram o impacto que a inclusão desse novo processo causou na utilização do processador pelos demais processos. Outros testes também estão detalhados nas linhas subseqüentes.

Nas linhas entituladas variação, pode-se verificar que a inserção de um novo processo afeta os demais processos de forma semelhante, não importando a sua prioridade. No entanto, seria de se esperar que os processos de maior prioridade sofressem um impacto menor na utilização do processador quando o novo processo fosse inserido, ao mesmo tempo que os de menor prioridade absorvessem mais o impacto da inserção. Embora em valores relativos todos os processos tenham impacto similar, em valores absolutos o processo mais afetado é o de maior prioridade, sendo ele o que mais cede tempo para que o novo processo possa ser executado.

O mesmo experimento foi realizado no sistema operacional Windows XP com Service Pack 2, com resultados que não puderam ser analisados adequadamente, pois este sistema operacional trata as prioridades de uma forma um tanto quanto inusitada. Em qualquer situação em que se tinha um processo com maior prioridade, sendo que outros processos possuíam prioridades diversas, o resultado era sempre muito similar. O gráfico 3.1[2]

[1]Os resultados nesse teste foram obtidos em uma estação Athlon XP 2600+ com 256MBs de memória rodando o Slackware Linux 10, com a versão 2.4.26 kernel

[2]Os testes foram realizados em um computador Athlon XP 2600 com 512MBs de memória. Foi utilizada a ferramenta *top* compilada para Windows [Noe98] para aferição dos resultados.

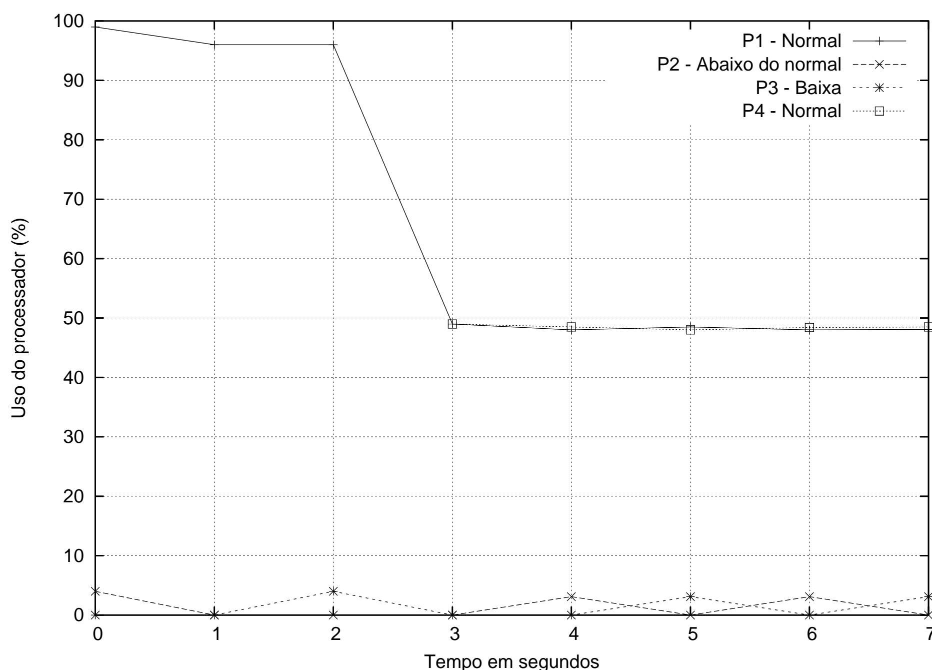

Figura 3.1: Uso do processador por N processos no sistema operacional Windows XP

apresenta o resultado obtido neste experimento, mostrando a inserção do processo P4, e concorrendo pelo uso do processor com os processos P1, P2 e P3, sendo que estes possuem diferentes prioridades.

Nota-se que o processo que mais transferiu tempo de uso da CPU para o novo processo foi aquele que possuía maior prioridade. Em outros testes, verificou-se que possuindo um processo de alta prioridade e outros de menor prioridade, ao inserir um novo processo com prioridade também menor, todos os processos continuavam com o mesmo tempo de execução no período que obtinham a CPU, no entanto demoravam mais períodos de tempo para receber novamente o processador.

Mesmo assim, com os resultados obtidos nos testes aqui descritos, fica fácil perceber que a solução adotada pelos sistemas operacionais de mercado não distribui justamente a utilização dos recursos entre os processos, já que o impacto do uso do recurso em alguns casos foi ainda maior nos processos de mais alta prioridade, sendo que o esperado fosse que sempre os processos de maior prioridade sofressem menos impacto em valores percentuais do que os de menor prioridade.

Além disso, os parâmetros usados para a decisão de escalonamento por muitas vezes não são os melhores para uma aplicação específica, ou para o sistema como um todo, ou ainda para a organização ou para os usuários.

Ademais, o administrador do sistema pode desejar que o processo de maior relevância para ele ou para a organização tenha acesso garantido a uma parcela significativa de um determinado recurso, mesmo com uma grande quantidade de processos concorrendo com ele, para satisfazer necessidades específicas de sua instituição.

No modelo atual de gerencimento de recursos adotado nos sistemas operacionais de mercado isso não é possível. E essa restrição de controle não ocorre somente para o processador, mas para a maioria dos recursos computacionais.

O problema do controle dos recursos é ainda mais complexo considerando que muitos recursos são interdependentes, ou seja, para que um recurso funcione adequadamente é necessário a utilização de outro recurso. Por exemplo, um gravador de CD, ao fazer uma gravação, necessita que os dados a serem armazenados sejam enviados a ele rapidamente para que a gravação não seja interrompida, o que faria o CD ser inutilizado. Para que os dados cheguem suficientemente rápido ao gravador é necessário que o meio que está fornecendo os dados para a gravação possa transferí-los de forma satisfatória. Caso o sistema operacional não forneça o acesso aos dois rapidamente, a gravação pode não ter sucesso.

Por melhor que seja a implementação do controle de recursos do sistema operacional, é impossível para ele conhecer quais são os parâmetros externos que devem ser analisados no agendamento dos processos. Por isso, é necessário que o administrador de sistema possa definir regras que indicarão ao sistema operacional quais as melhores escolhas a serem feitas no momento de fornecer o acesso aos recursos.

Essas regras podem também ser usadas quando se deseja que alguns processos obtenham mais freqüentemente um recurso, ou quando se quer restringir recursos para algumas tarefas menos relevantes para a organização, liberando por mais tempo os recursos para os processos mais importantes. O modelo de prioridades tenta realizar essa tarefa, mas não a cumpre por completo, já que mesmo tendo prioridade baixa o processo pode obter até 100% de uso de um recurso caso não haja nenhum outro processo concorrendo com ele.

Essas restrições de acesso aos recursos também podem ser aplicadas a processos maliciosos, processos sob ataques de negação de serviço ou processos afetados por intrusões. Para tanto, aplicativos de detecção de intrusão, como o Snort [Roe99], podem ser usados para detectar a invasão e os processos comprometidos, e, assim, interagindo com o sis-

tema de controle dos recursos, lançar restrições sobre eles, a fim de manter a integridade do sistema. Ao evitar simplesmente matar o processo comprometido, mantém-se o valor forense das informações contidas no processo para que possam ser analisadas por peritos, responsabilizando os culpados.

Tudo isso é possível com um modelo de controle de recursos que permita que o administrador do sistema, ou os próprios usuários, interajam com o mecanismo de controle de recursos. Assim, processos relevantes para a organização ou para o sistema podem ser privilegiados na obtenção de recursos devido à sua importância. Bem como, processos de outras naturezas, como maliciosas ou não fundamentais, podem ser restritos.

Os problemas descritos aqui foram percebidos e por isso algumas tentativas de melhoramento do controle dos recursos já foram propostas e são apresentadas na próxima seção.

3.3 Mecanismos avançados de gestão de recursos

Nessa seção serão descritos alguns métodos alternativos de gerência dos recursos computacionais que visam suplantar os métodos convencionais.

3.3.1 Limits

O padrão System V e BSD definem um mecanismo para controlar os recursos do sistema. A biblioteca `libc` implementa esse mecanismo e o disponibiliza para ser usado em sistemas Unix.

As funções `setrlimit` e `getrlimit` permitem respectivamente definir e obter os limites dos recursos que esse mecanismo é capaz de controlar [Pro99a]. Com o `setrlimit` é possível definir dois tipos de limites: o *soft* e o *hard*. O primeiro é utilizado como um alerta ao processo que o limite permitido foi atingido. E o último funciona como limite máximo permitido para utilização do recurso, sendo que a partir do momento que é atingido, o sistema toma providências para voltar a situação de satisfação do limites, inclusive matando o processo que utiliza os recursos indevidamente. É bom salientar que esses limites são aplicados exclusivamente por sessão de usuário e não por processo.

Os recursos controlados por esse mecanismo são:

- RLIMIT_CPU (processador): especifica o limite máximo em segundos em que um processo pode utilizar o processador. Quando o processo atinge o limite *soft* um sinal de término é enviado ao processo. Se o processo ainda não estiver terminado quando o limite *hard* for atingido o processo será morto;
- RLIMIT_CORE (tamanho do arquivo de *core*)[3]: ajusta o tamanho máximo do arquivo de *core*. Quando seu valor é 0 (zero), o arquivo de *core* não é criado;
- RLIMIT_DATA (segmento de dados): limita o tamanho máximo do segmento de dados em memória utilizados pelos processos;
- RLIMIT_LOCKS (número de arquivos bloqueados): define o limite máximo de arquivos que podem ser bloqueados ao mesmo tempo através das funções `flock()` e `fcntl()`;
- RLIMIT_MEMLOCK (bytes alocados pelo usuário em memória física): permite especificar a quantidade de bytes que ficarão em memória física e não serão colocados na memória virtual;
- RLIMIT_FSIZE (tamanho dos arquivos): limita o tamanho máximo dos arquivos que podem ser criados pelos processos;
- RLIMIT_NOFILE (número de arquivos): limita o máximo de descritores de arquivos que podem ser abertos pelos processos;
- RLIMIT_NPROC (número de processos): ajusta o número máximo de processos simultâneos de um usuário;
- RLIMIT_RSS (páginas na memória): limita o máximo de páginas que um processo pode manter residente na memória física;
- RLIMIT_STACK (tamanho da pilha): especifica o tamanho máximo da pilha de cada processo.

Mais detalhes podem ser encontrados no Manual do Programador Linux [Pro99a]. Pode-se notar que esse mecanismo permite definir somente o limite máximo de utilização

[3] Arquivo gerado pelo núcleo que contém a imagem da memória de um processo abortado devido a problemas (falha de segmentação, etc.) Pode ser utilizado por desenvolvedores para depurar programas problemáticos.

dos recursos, não permitindo garantir limites mínimos de uso. Além disso, são definidas por sessão de usuário e não são alteráveis durante a vida dos processos.

3.3.2 Class-Based Kernel Resource Management — CKRM

O CKRM é um gerenciador de recursos orientado a objetivos que visa fornecer serviços diferenciados a classes de tarefas para todos os recursos que ele pode gerenciar [NvRF+04a].

Uma classe é definida como um grupo dinâmico de objetos do sistema operacional de um tipo específico, como processos e *threads*. As classes possuem compartilhamentos associados aos recursos do sistema [NvRF+04b], que são definidos pelo administrador do sistema. Cada classe obtém compartilhamentos separados para tempo de CPU, páginas de memória, banda de disco e banda de rede. O escalonador do recurso tenta da melhor forma possível respeitar os limites dos compartilhamentos da classe.

Por exemplo, uma classe A com um compartilhamento de 50% de tempo de CPU, coletivamente obtém 50% do uso do processador [NFC+03], ou seja, para todos os processos da classe.

O CKRM é dividido em vários componentes [NvRF+04a]:

- Core: define as entidades básicas usadas pelo CKRM, como as classes e os eventos, e atua como elo de ligação entre todos os outros componentes;

- Método de classificação: é um componente opcional e ajuda na associação automática dos objetos do núcleo às classes de seu tipo. Atualmente existem dois tipos de classe: de tarefas e de sockets, para o agrupamento de processos e sockets respectivamente.

- RCFS (sistema de arquivos de controle de recursos): provê uma interface de criação e manipulação das classes baseado em diretórios e arquivos. A comunicação entre o usuário e núcleo é feita através de leituras e escritas em arquivos nos diretórios do sistema de arquivos. Os diretórios do RCFS corresponde às classes.

- Controladores de recursos: são os reguladores de utilização dos recursos. São implementações independentes e estão interessadas somente no controle de um recurso específico, ou seja, existe um controlador para cada recurso que se deseja gerenciar.

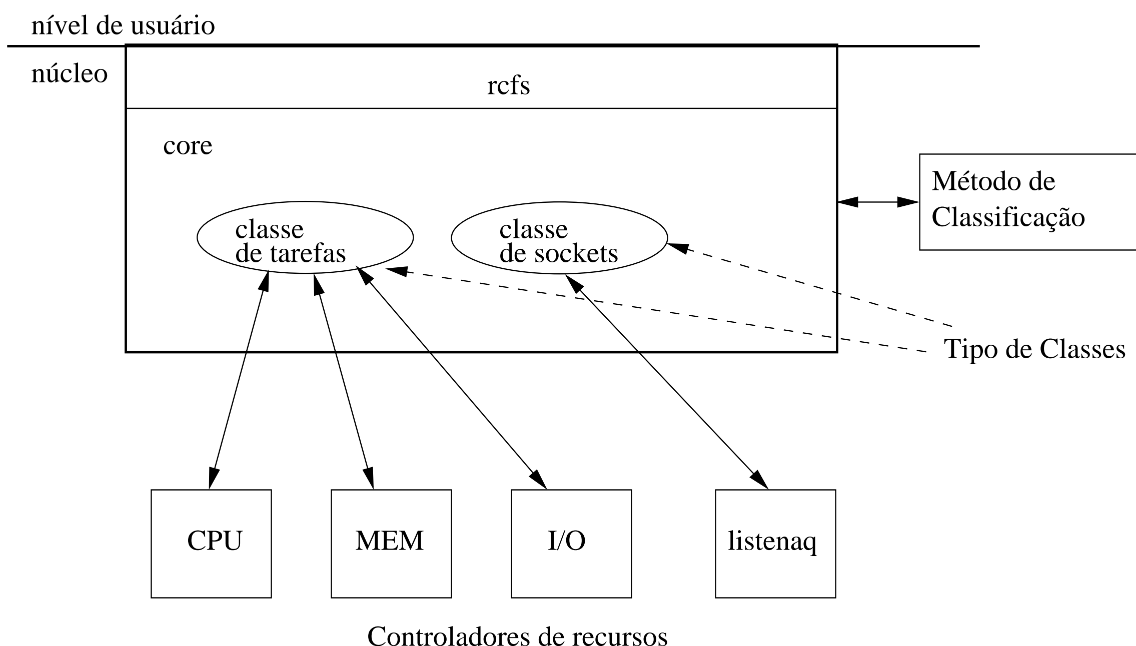

Figura 3.2: Projeto de implementação do CKRM.

O fato de ser dividido em módulos permite que pessoas interessadas em construir um novo controlador para algum recurso possam utilizar a estrutura fornecida pelo CKRM, diminuindo consideravelmente o tempo de desenvolvimento.

A figura 3.2 [SNK05] apresenta o modelo de implementação do CKRM, os seus componentes e como estão relacionados entre si.

O CKRM também permite um melhor mensuramento do uso do sistema. Atualmente é difícil atribuir os recursos consumidos em *kernel mode* a uma determinada tarefa. Com uma medida mais acurada do consumo dos recursos, o administrador do sistema pode decidir mais precisamente no controle de uma aplicação em particular ou conjunto de tarefas no sistema.

3.3.3 Linux-SRT

O Linux-SRT é uma versão aprimorada do Linux padrão que implementa uma política de escalonamento de tempo-real flexível (*sotf real-time*). Esse sistema pode fazer a alocação de recursos, garantindo um limite mínimo de utilização. Esse limite pode não ser alcançado devido a algum fator, no entanto, a alocação real obtida estará muito próxima da especificada.

Como a alocação de tempo de processamento não é suficiente para garantir o compor-

tamento previsto para a aplicação, esse modelo provê, além de um novo escalonador de uso do processador, uma nova política de agendamento de disco [CI01].

O Linux padrão possui três políticas de escalonamento de processos para utilização da CPU: prioridades, FIFO e tempo-real. O Linux-SRT habilita mais três políticas de escalonamento. A primeira permite que as aplicações especifiquem uma quantidade fixa de uso do processador em um determinado período. A segunda é utilizada para manter uma aplicação suspensa, enquanto a última define que uma aplicação só pode executar quando o processador estiver ocioso.

As mesmas políticas foram implementadas para o agendamento do uso da banda de disco. Foram criadas mais três políticas, uma onde a aplicação define uma quantidade fixa de uso da banda, outra que fará com que as requisições de acesso a disco sejam postergadas e a última que permite o processo utilizar a banda de disco somente quando ela estiver ociosa.

A contribuição do Linux-SRT é combinar escalonadores de vários dispositivos, resultando em um sistema com gerenciamento de QoS realmente integrado.

A unidade de alocação dos recursos é chamada de *reserve*, e é um conjunto de processos definido pelo usuário. Somente um processo pode ser incluído no *reserve* por vez.

É possível definir tanto uma quantidade mínima quanto máxima de utilização dos recursos, no entanto, os ajustes só podem ser realizados em tempo específico de configuração.

3.3.4 Dynamic Soft Real-Time CPU Scheduler

O sistema DSRT — Dynamic Soft Real-Time CPU Scheduler [Jac02] — implementa um escalonador de uso do processador em nível de usuário (ou seja, fora do núcleo) capaz de garantir tempo de processamento em aplicações de tempo-real que suportem pequenos atrasos (flexíveis).

Por ser um escalonador de tempo-real flexível, esse modelo tenta cumprir os prazos determinados em contratos de QoS, mas pode ser influenciado por interrupções ou faltas de página. O objetivo desse modelo é prover um mecanismo para permitir que aplicações multimídia de tempo-real possam ser executadas adequadamente.

As requisições de alocação de CPU são feitas utilizando 3 (três) argumentos: a quan-

tidade de reserva do recurso, a duração da reserva e o momento do início da reserva. Não sendo possível garantir o uso do processador de acordo com o estipulado pela requisição, o sistema oferece a possibilidade de diminuir a reserva, ou diminuir a duração da reserva ou mesmo alterar o início da reserva.

A arquitetura do DSRT mostra que ele se ajusta bem para aplicações multimídia que, geralmente, possuem tempo de duração bem determinado, necessitam de acesso privilegiado aos recursos e toleram o não cumprimento exato do contrato de QoS.

Por se tratar de um escalonador que executa em *user mode*, a sua portabilidade é facilitada, tanto que existem versões do DSRT para Linux, SunOS, Windows NT e Irix. Não há necessidade de alterações no núcleo do sistema operacional para implementá-lo.

Infelizmente esse sistema não é granular, ou seja, trata somente aplicações que necessitam ser modificadas para que possam utilizar o mecanismo de reserva de recursos implementado pelo DSRT. É importante salientar que esse sistema não é dinâmico, pois é necessário informar os requisitos de execução no lançamento do processo.

Não há suporte para a definição de limites máximos de uso do processador.

3.3.5 Cap Processor Usage

Também chamado de CPUCap, consiste de um *patch* para o núcleo do Linux que permite limitar o quanto um processo pode utilizar o processador [Gol05].

Esse modelo de gerência de recursos não cria uma nova política no núcleo do Linux, somente altera a política dos processos comuns, adicionando uma verificação que torna possível a aplicação de limites máximos de utilização da CPU para algum processo específico.

A tabela de controle de processos do Linux também é alterada, sendo adicionada a ela informações sobre os limites de uso do processador para aquele processo. A granularidade desse sistema é o processo, não permitindo que limites sejam impostos sobre qualquer tipo de agrupamento de processos. Não é possível definir limites inferiores para os processos. Há pouca documentação disponível, e no momento a versão mais atual é para o núcleo do Linux 2.6.6.

Tabela 3.2: Diferenças entre os modelos alternativos de gestão de recursos.

	Limits	CKRM	Linux-SRT	DSRT	CPUCap
Dinâmico	Não	Sim	Não	Não	Sim
Limite Superior	Sim	Sim	Sim	Não	Sim
Limite Inferior	Não	Sim	Sim	Sim	Não
Granularidade	Sessão	Classe	Classe	Aplicação	Processo
Recursos	CPU Memória Internos	CPU Memória Disco Rede Outros	CPU Disco	CPU	CPU

3.4 Conclusões

Conforme pôde ser visto, existem vários métodos para limitar o uso dos recursos. O problema é que cada um implementa somente algumas das características requeridas para um modelo de gestão de controle de recursos. Por exemplo, alguns deles só limitam o máximo do uso, ou seja, pode-se restringir o acesso aos recursos, o que pode ser requerido para diminuir a eficiência de um programa sem, no entanto, priorizar outros.

A tabela 3.2 mostra as diferentes características dos modelos apresentados na seção anterior.

O CKRM é o modelo apresentado com mais funcionalidades. Como foi mostrado, ele implementa as funções mais desejadas em um método de gerência de recursos.

Poucos ainda são os sistemas que garantem o uso mínimo dos recursos para os processos, isto é, um sistema que priorize processos específicos independentemente se os outros vão ter o acesso aos recursos restringidos em conseqüência disso. Os sistemas que geralmente implementam essa característica são aqueles que visam garantir QoS. Esses sistemas não tem como prioridade a implementação do gerenciamento dos recursos, mas o desempenho de uma aplicação específica.

No próximo capítulo será proposto um modelo para gerenciar os recursos de forma que tanto limites máximos e mínimos do uso dos recursos possam ser incorporados ao sistema e ajustados de forma dinâmica para atender necessidades específicas.

Capítulo 4

Um modelo de gestão de recursos

Neste capítulo será proposto um modelo dinâmico de gestão de recursos que, como será visto, se diferencia dos modelos convencionais atuais por sua melhor adaptabilidade às necessidades específicas dos usuários e pela capacidade de responder instantaneamente às modificações feitas nos parâmetros utilizados para alocação dos recursos para os processos.

4.1 Hierarquia do uso dos recursos

Em um sistema computacional, os processos em execução obtêm acesso aos recursos físicos e lógicos do computador através das designações feitas pelo sistema operacional. Para que haja uma justa distribuição da utilização dos recursos é necessário que o sistema operacional gerencie o uso de cada um dos recursos. Assim, a responsabilidade de escolher os processos que terão as suas requisições de acesso aos recursos satisfeitas é do sistema operacional. Essa escolha é feita por um algoritmo que avalia qual a melhor alternativa de alocação dos recursos para um melhor desempenho do sistema ou para satisfazer prioridades impostas pelos usuários.

Para cada recurso existem políticas de decisão de alocação e agendamento dos recursos destinados aos processos [SGG02]. São essas políticas que determinam a ordem de utilização dos recursos. Como exemplo tem-se a política de utilização dos processadores em sistema Unix que priorizam processos que fazem maior uso dos recursos de entrada e saída, ao invés de priorizar os processos orientados ao uso do processador. Esta política é utilizada devido ao fato que, em geral, processos que utilizam mais freqüentemente os dispositivos de entrada e saída são interativos, ou seja, aqueles que o usuário está fornecendo

dados e, na maioria das vezes, esperando a resposta no mesmo momento.

Para possibilitar as tomadas de decisões feitas pelas políticas de escolha, existem hierarquias às quais os processos estão submetidos. A hierarquia é definida por algum parâmetro, tal como prioridade, proximidade física, uso dos recursos, ordem de requisição e outros fatores.

Para tentar melhorar o desempenho de uma aplicação específica, o administrador pode, por exemplo, aumentar a prioridade deste processo, fazendo que o mesmo obtenha o processador com mais freqüência. Mas, mesmo alterando a prioridade do processo, isso não garante que a aplicação terá um aumento significante no seu desempenho, já que esse processo pode ter que esperar por outros recursos que ainda não estão disponíveis a ele, visto que esses outros recursos podem possuir políticas de decisão diferentes.

A debilidade de um sistema em aumentar o desempenho de uma aplicação consiste, então, na falta de políticas globais de priorizar ou restringir o acesso aos recursos.

É nítido que por muitas vezes é necessário modificar as políticas não somente de um processo, mas também de todos os processos de um usuário ou grupo de usuários. Isso se deve ao fato de que determinados usuários realizam atividades de maior relevância que outros. Também pode-se modificá-las de forma a afetar um grupo de processos relacionados, como os processos de uma mesma aplicação, devido à sua importância para a organização.

Melhor seria, então, que as políticas de decisão de alocação e agendamento de recursos fossem tomadas também com base nos recursos destinados a uma *thread*, a um processo, a um grupo de processos, a um usuário ou um grupo de usuários. Como os sistemas Unix já implementam essas unidades, é natural a possibilidade de definir que elas sejam usadas também como forma de gerenciar os recursos.

Esse capítulo visa apresentar um modelo que fornece ao administrador políticas para a alocação de recursos de forma que as atividades realizadas no sistema possam ser privilegiadas ou restringidas na utilização dos mesmos.

4.2 O controle dinâmico do uso dos recursos

A proposta deste modelo é definir um método de controle dinâmico de recursos para um sistema operacional multi-usuário. Por controle entende-se a possibilidade de definir

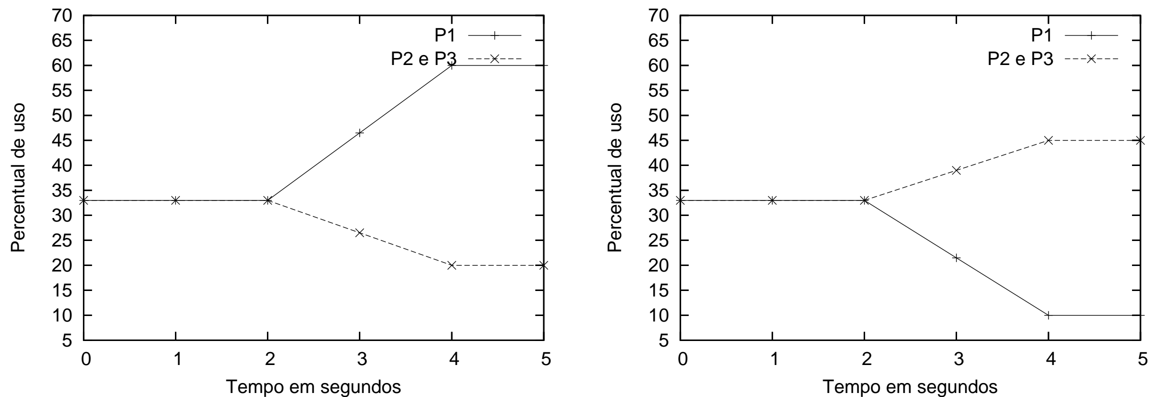

Figura 4.1: Aplicação de limite inferior (esquerda) e superior (direita) no uso do processador.

limites inferiores e superiores na disponibilidade de recursos aos processos existentes no sistema. Esses limites podem ser estabelecidos e reajustados dinamicamente, em qualquer momento da vida das atividades que serão controladas, não necessitando suspendê-las ou reiniciá-las para a aplicação dos limites.

O limite superior consiste da máxima utilização permitida do recurso, enquanto que o limite inferior garante o aproveitamento mínimo do recurso caso esse seja necessário. Os diagramas da figura 4.1 exemplificam os conceitos de limite inferior e superior no uso dos recursos. No diagrama à esquerda, três processos (P1, P2 e P3) com as mesmas características e mesma prioridade disputam o tempo de processador, ficando cada um com 33% do mesmo. Ao ajustar o limite inferior de P1 para 60% em $t = 2$, este passa a usar 60% do tempo de processador, enquanto os outros 40% do recurso são divididos entre P2 e P3. No caso do diagrama à direita ocorre o inverso: o limite superior de tempo de processamento de P1 é ajustado para 10%; neste caso, P2 e P3 dividem entre si o percentual restante (90%).

Nos diagramas da figura 4.1 observa-se que a mudança na distribuição do recurso entre os processos não ocorre imediatamente. Devido à natureza dos algoritmos de escalonamento dos recursos em jogo (processador, memória física e outros recursos), existe um período de transição durante o qual o sistema deve se adequar às novas restrições. Um dos objetivos da proposta é minimizar a duração desse período de transição.

Nesse modelo, a política de decisão é feita através de regras de alocação de recursos que são definidas pelo administrador do sistema. Essas regras definem quais os recursos poderão ser acessados por quais processos e dentro de quais limites.

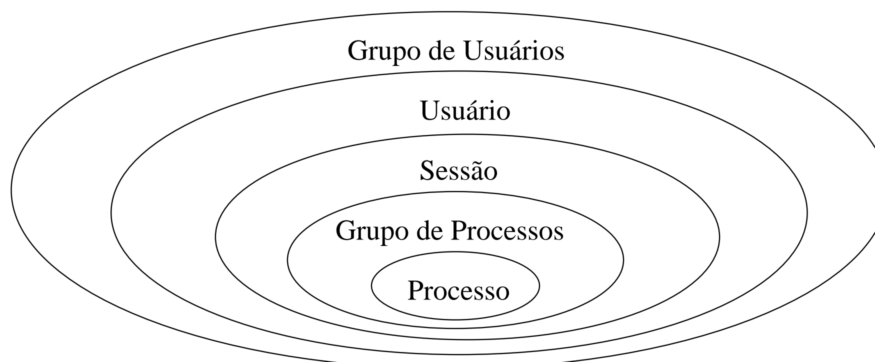

Figura 4.2: A granularidade do acesso aos recursos.

Normalmente o controle do uso dos recursos é feito com base em processos, visto que estes são a unidade básica de alocação de recursos nos sistemas operacionais correntes[1]. Algumas vezes, no entanto, pode ser necessário definir regras de alocação de recursos para grupos de processos relacionados. Assim, a definição de regras de alocação de recursos deve considerar diferentes níveis de agrupamento de processos. Os níveis considerados na presente proposta são apresentados na figura 4.2.

O grupo de processos indicado na figura 4.2 é uma coleção de processos que possuem um identificador de grupo (`group id`) em comum. Esse identificador é fornecido pelo processo chamado líder do grupo. Qualquer processo pode tornar-se líder, bastando executar a chamada de sistema apropriada. Tal ação gerará um novo identificador de grupo e o tornará líder de seu próprio grupo [Lib01].

Analogamente, uma sessão de processos define um conjunto de grupos de processos que compartilham um mesmo identificador de sessão (`session id`), pertencente originalmente ao líder da sessão. Para um processo tornar-se líder de sessão basta executar a chamada de sistema apropriada, gerando um novo identificador de sessão.

A política de acesso às regras de definição de limites é feita com base no usuário que ativa a regra. Um usuário pode restringir o acesso aos recursos de seus próprios processos, ou seja, ele pode definir regras aplicáveis sobre processos, sessões ou grupos de processos de sua propriedade, contanto que sejam regras estabelecendo limites superiores no uso dos recursos. Somente o administrador do sistema poderá criar ou alterar regras definindo limites inferiores, ou regras sobre usuários ou grupos de usuários.

[1] Isso pode variar dependendo da natureza do recurso em questão. Por exemplo, enquanto a unidade básica de alocação de memória é o processo, em núcleos que implementam modelos de *threading* $1-1$ ou $M-N$ a unidade básica de alocação de processador é a *thread*, e não o processo.

Deve-se ter em mente que, ao definir regras de limite inferior, somente a reserva de recursos é feita. A sua utilização efetiva dependerá do processo (ou grupo) ao qual a regra se aplica, pois o núcleo do sistema somente pode garantir que o recurso estará disponível caso seja necessário, e não pode forçar o processo a efetivamente utilizar aquele recurso caso não o necessite.

É de se esperar que, ao se tratar regras independentes, um processo possa ser afetado por duas ou mais regras, que podem, inclusive, restringir e privilegiar ao mesmo tempo para um mesmo recurso. Quando isso ocorrer, a regra mais restritiva será a utilizada.

A seguir será proposto o modelo de controle dinâmico para os principais recursos computacionais.

4.2.1 Processador

O processador é um dos recursos que possuem métricas relativas. A utilização do processador é medida pelo tempo que o processo o utiliza. Para que seja possível fazer esse controle é necessário delimitar um espaço de tempo, conhecido como ciclo, onde serão feitas as quantificações de utilização.

O diagrama da figura 4.3 mostra o uso do processador pelo processo P1 durante vários ciclos. A duração de cada ciclo é definida como sendo 1 (um) segundo.

Geralmente é através das prioridades dos processos e de compartilhamento de tempo que os sistemas operacionais modernos implementam o mecanismo de compartilhamento do uso do processador. Comumente são definidos pequenos períodos de tempo, menores que um ciclo, chamados *quantum*, que correspondem ao período de tempo de execução de um processo durante o ciclo. A escolha de qual processo irá utilizar o processador é feita através da prioridade do processo e de quanto ainda resta de seu *quantum*.

Esse mecanismo não consegue ser restritivo pois só estabelece prioridades na execução do processo. Se não existe disputa para a obtenção do processador, um processo de baixa prioridade ainda irá obter 100% de uso do processador, o que pode não ser o desejo do administrador do sistema. Da mesma forma que também não é possível garantir a utilização mínima do processador por um processo ou grupo de processos, pois caso a concorrência pelo uso do processador seja grande, o sistema operacional tentará balancear a utilização do processador entre os processos, de acordo com suas prioridades.

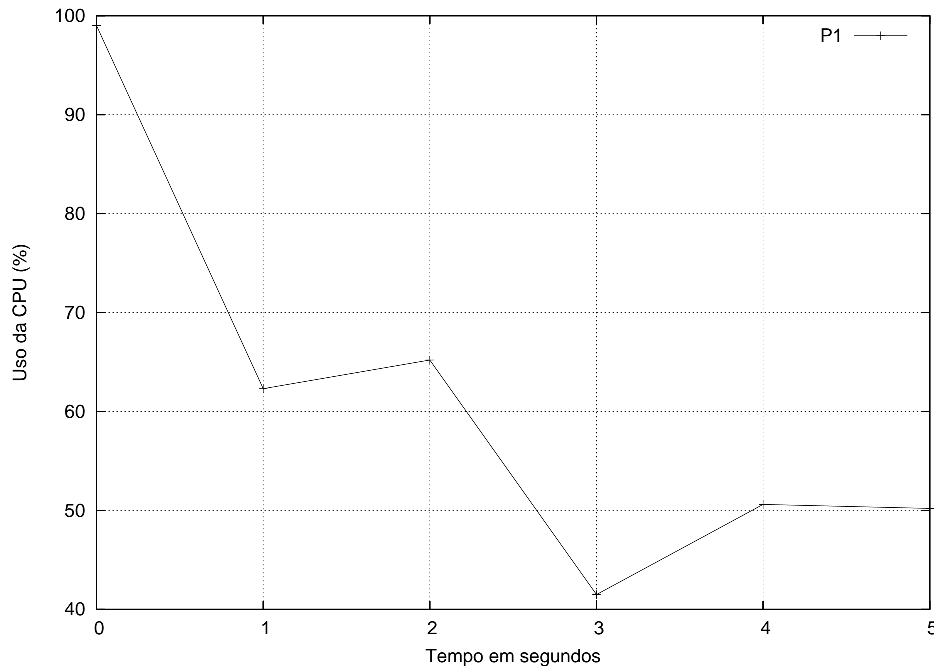

Figura 4.3: Utilização do processador por P1 durante N ciclos.

O que se propõe para o controle do uso do processador nesse modelo é uma nova política pela qual o sistema operacional determina a utilização do processador através de regras definidas pelo administrador do sistema. Essas regras definem em valores percentuais a utilização do processador durante um ciclo por um processo ou grupo de processos.

Para que isso seja possível é necessária a definição de um novo método de escalonamento de processos. Essa política estabelece que um processo deve deixar de usar o processador se:

- um processo de tempo-real torna-se apto a executar;
- o percentual de utilização do processador seja atingido, se a regra define um limite superior, não podendo novamente obter o processador nesse ciclo;
- o percentual de utilização do processador for ultrapassado, se a regra define um limite inferior e há outros processos esperando para execução[2];

[2] É necessário que o processo que possui um limite mínimo libere o processador a fim de não monopolizá-lo, e assim, outros processos possam também ser executados. Caso não haja outro processo na fila de execução, pode haver continuidade na execução deste mesmo processo, no entanto, a partir deste momento, ele concorre pelo uso do processador da mesma maneira que processos não afetados por regras o fazem.

- o processo termine ou seja bloqueado em uma operação de entrada e saída ou de sincronização.

Em outras palavras, um processo ou grupo de processos que são afetados por uma regra de limite inferior devem ser executados até que o percentual especificado pela regra seja ultrapassado. Após esse evento ocorrer, eles podem concorrer com os outros processos normalmente. Para um grupo de processos o percentual de utilização é definido pela soma do uso do processador de todos os componentes do grupo durante o ciclo.

Já uma regra de limites superior é satisfeita se o processo ou grupo de processos a que ela se refere não ultrapassar o percentual de uso definido na regra. Se se trata de uma regra de grupo de processos, a utilização do processador é encontrada através da soma do uso do processador pelos integrantes do grupo.

Os processos que não são pertinentes a nenhuma regra continuam sendo escalonados na forma usual do sistema operacional em questão. Não são necessárias mudanças no mecanismo convencional de escalonamento.

Com esse método implementado é possível definir políticas de decisão de alocação do processador e escalonamento de processos segundo o percentual de uso da CPU. O administrador pode tanto criar regras de limites superiores quanto inferiores de forma que os processos com menos ou mais relevância sejam executados de maneira quantificada.

No momento da definição de regras de limite inferior, deve-se verificar que a soma dos limites de todas as regras de limite inferior não ultrapasse a quantidade máxima de utilização do recurso. Por exemplo, não é possível criar regras definindo limites inferiores de uso de processador nas quais a soma dos limites seja superior a 100%. Portanto, regras que excederem a capacidade disponível do recurso em questão devem ser descartadas.

É bom notar que, ao utilizar regras de limites inferiores, o sistema primeiro executa os processos pertencentes à regra, não liberando o processador para os outros processos até que eles tenham utilizado todo o tempo ao qual eles possuem direito. Assim sendo, ao se ajustar 70% de utilização do processador para um processo, ele executará por 700ms antes de deixar a CPU. Isso pode não ser desejável em alguns sistemas onde essa quantidade de tempo sem executar outro processo seja inaceitável. Essa característica do modelo é alvo de trabalhos futuros, onde o sistema possa detectar se ainda há tempo suficiente para executar os processos da regra, e liberar a CPU para outros processos antes que a regra

tenha sido satisfeita, ao mesmo tempo que garanta o cumprimento dela.

4.2.2 Memória

Todos os recursos possuem mecanismos que hierarquizam os processos a fim de permitir o uso adequado. A memória, no entanto, possui mecanismos pouco configuráveis que, na maioria das vezes, não permitem a interferência do administrador.

Com o intuito de permitir que o administrador possa interagir na política de decisão de uso da memória, esse modelo prevê três abordagens para o controle desse recurso:

- Quantidade de páginas do processo (ou grupo) residentes em memória física (RSS - *resident set size*);
- O número de faltas de página que ocorre;
- Quantidade de páginas em memória virtual.

O primeiro especifica quantos quadros de páginas um processo (ou grupo) pode utilizar, o que equivale a dizer quantos KBs o processo pode utilizar da memória RAM. O segundo especifica com que freqüência ocorrerá as faltas de página para o processo (ou grupo) e o último define o limite de páginas que podem estar residentes na memória virtual.

Memória física

Como já foi dito, um processo tem seu código e seus dados armazenados em páginas, que podem estar tanto alocadas na memória física quanto em disco. A quantidade de páginas que o processo possui alocada em memória física é conhecida como RSS, ou tamanho do conjunto residente.

O modo de controle, quando utilizando esse método para tratar as regras de limite inferior e superior, é limitar o valor mínimo e máximo, respectivamente, que o RSS pode atingir. Esse recurso é considerado preemptível, visto que se uma página de processo não tem o direito de permanecer em memória física ela pode ser transferida para a área de *swap* sem corromper a execução do processo.

As regras de limite superior delimitam o RSS máximo para o processo (ou grupo). Assim, não é permitido ao processo manter mais páginas em memória física do que o

especificado pela regra. Caso isso ocorra ele terá algumas de suas páginas transferidas para o *swap* a fim de satisfazer a condição da regra.

Para regras de limite inferior o que é especificado é o número mínimo de páginas alocadas em memória física para o processo (ou grupo). Isso significa que se o processo possuir páginas suficientes para ocupar todo o valor estipulado pelo limite inferior, elas ficarão alocadas em memória física. Porém, se o processo não possuir páginas suficientes, o recurso será utilizado por outros processos até que sejam requeridas novas alocações de página pelo processo (ou grupo) controlado por esse tipo de regra.

Deve-se notar que a soma dos valores de todas as regras de limite inferior não pode ultrapassar o número máximo de quadros disponíveis no sistema. Caso isso ocorra a regra deve ser descartada.

Freqüência das faltas de página

Uma falta de página ocorre quando um processo que está sendo executado possui uma das páginas necessárias para sua execução alocada em *swap*. A falta de página ocorre para informar ao sistema operacional que uma página necessária para a execução do processo se encontra em disco, e dessa forma, ele possa transferí-la para a memória física.

Um número grande de faltas de página pode significar que uma ou mais páginas de memória que um processo usa constantemente estão sendo transferidas para a área de *swap* pela necessidade que o sistema tem de utilização de memória física por outros processos devido a escassez de memória livre. Também pode significar que um processo seja demasiadamente grande e está fazendo uso de várias páginas diferentes que ainda estão em disco. Impor limites sobre a quantidade de faltas de página que ocorrem traduz-se por manter um determinado número de páginas ou em memória física ou em *swap* de acordo com o que especificar a regra.

Para quantificar a freqüência de faltas de página deve-se encontrar a relação do número de faltas de página, que ocorrem para um ou mais processos, com um determinado período. Como não é possível saber quantas faltas de página ocorrerão em um período de tempo pré-definido, visto que não se pode prever a necessidade de uso das páginas, deve-se utilizar uma outra maneira de estipular um período. Neste modelo foi utilizado uma quantidade c de faltas de página ocorridas como período. Assim sendo, um processo tem a sua freqüência de faltas de página medida pelo número de faltas de página que ele gerou

durante c faltas de página ocorridas no sistema. Por exemplo, se $c = 100$, um processo P1 que gerou 15 faltas de página durante as 100 últimas faltas ocorridas no sistema todo, tem 15% de freqüência de faltas de página. Neste modelo, as faltas de página do sistema devem ser contadas na forma de ciclos de tamanho c.

O limite inferior de uma regra de controle da freqüência das faltas de páginas especifica que de todas as faltas de página ocorridas no sistema durante o período c, o processo (ou grupo) a qual a regra se refere pode no máximo gerar a quantidade de faltas de página especificado pela própria regra. Um método para se implementar isso é manter em memória física uma quantidade suficiente de páginas do processo (ou grupo) pertinentes a regra de forma que se ele necessitar de algum dado ou instrução eles já estejam em memória física, e se estiverem em *swap*, a quantidade de páginas transferidas não vai ser suficiente para ultrapassar o valor especificado pela regra neste ciclo c.

O algoritmo para substituição das páginas pode ser o mesmo utilizado pelo sistema operacional hospedeiro. Assim sendo, caso a página retirada seja de um processo pertencente a regra, pode-se acarretar no descumprimento da regra se a página retirada for requerida novamente antes de se completar o ciclo. No entanto, um algoritmo que não permita a retirada dessa página da memória se tornaria um simples método de manter todas as páginas do processo na memória, o que pode não ser desejado.

Já o limite superior determina a quantidade máxima de faltas de página que os processos pertencentes à regra podem gerar durante o período c. Caso os processos atinjam o limite de faltas de página permitido, eles devem esperar um novo ciclo c para continuarem a sua execução, ficando bloqueados. É fato que se houver demora para a conclusão do ciclo, os processos bloqueados devido a regras de limite superior podem também demorar para tornarem-se prontos para execução novamente. Assim sendo, é bom que o ciclo expire após um tempo pré-determinado.

Os processos que não participam de nenhuma regra continuam a seguir a mesma política de decisão estipulada pelo sistema operacional.

Memória virtual

A utilização da memória virtual também pode ser controlada de forma que não se permita que um processo (ou grupo) possua uma quantidade de páginas em memória virtual maior do que especificado, ou que se garanta espaço de alocação para as páginas

do processo (ou grupo) mesmo quando a memória virtual esteja alcançando seu limite (memória física + área de *swap* saturadas).

As regras para este controle devem estipular valores absolutos do número mínimo ou máximo de páginas que podem ser alocadas na memória virtual.

Para as regras de limite superior, os processos que são alvo das regras não podem manter um número maior de páginas em memória virtual do que o imposto pela regra. Assim sendo, quando estes processos alocarem memória, o limite estipulado é verificado e, caso seja ultrapassado, o sistema não permite a nova alocação de memória e retorna um código de erro.

Se no momento da criação de uma regra de limite superior, os processos afetados pela regra já ultrapassam o limite fixado, deve-se permitir que os processos executem acima do limite, negando qualquer nova tentativa de alocação até que eles liberem espaço em memória virtual e o limite seja satisfeito.

Os processos que fazem parte de regras de limite inferior tem o privilégio de reserva de espaço na área de *swap* no momento da criação da regra de acordo com o valor definido por esta, se houver espaço livre suficiente. Caso não haja, a regra não é criada. As páginas em *swap* dos processos que já estão em execução no momento da criação da regra também devem ser contadas e marcadas como reservadas.

Ao transferir uma página da memória física para a área de *swap*, deve-se verificar se a página pertence a um processo de uma regra de limite inferior. Se o for, deve-se alocá-la no espaço previamente reservado.

4.2.3 Disco

Basicamente pode-se fazer dois tipos de controle sobre as unidades de disco:

- banda de disco;
- espaço de disco utilizado.

O controle da utilização da banda de disco estabelece o quanto de dados podem ser transferidos do sistema computacional para o disco e vice-versa por um processo ou grupo de processos.

O controle da utilização do espaço em disco geralmente destina-se a quanto um usuário ou grupo de usuários podem armazenar de dados em seus próprios diretórios, o que a maioria dos sistemas operacionais já implementa. No entanto, pode-se querer controlar o espaço de disco utilizado por um ou mais processos, que é o objetivo deste modelo de gestão de recursos.

Banda de disco

Na maioria dos sistemas operacionais de mercado a escolha de qual dado deve ser escrito ou lido primeiro depende somente da posição da cabeça de leitura e gravação sobre as superfícies magnéticas do disco. Isso se deve ao fato de minimizar o deslocamento da cabeça de leitura, o que resulta em um melhor desempenho do disco.

É claro que a eficiência na leitura e escrita em discos rígidos está relacionada também ao limite máximo da banda de disco imposta pelo hardware.

Para implementar um novo mecanismo de priorização de entrada e saída de dados em discos rígidos que considere as regras impostas pelo administrador, primeiramente deve-se conhecer qual a banda máxima do disco, levando em conta a sobrecarga do sistema de arquivos. Com isso, é possível aceitar limitações inferiores e superiores de banda de disco, tanto em termos de percentuais de uso como valores absolutos de utilização de banda. Como a banda é medida através do tempo, também aqui é necessária a definição de ciclos para que se possa mensurar o uso da banda de disco.

As regras não podem ultrapassar o limite físico da banda, logo os valores de utilização da banda de disco das regras devem estar entre 0 e 100%, sendo que o primeiro significa a não utilização do disco e o segundo a banda máxima de transferência. Como há diferenças de banda de disco de sistema para sistema, ou mesmo de disco para disco, deve-se avaliar qual é a banda física localmente, possivelmente na inicialização do sistema.

O novo mecanismo de agendamento de utilização do disco estabelece que as requisições de leitura e gravação serão satisfeitas na seguinte ordem:

1. se a requisição pertence a um processo ou grupo de processos que possuem uma regra de limite inferior que ainda não foi ultrapassada nesse ciclo;

2. se a requisição pertence a um processo ou grupo de processos que possuem uma regra de limite superior que ainda não foi atingida nesse ciclo.

Em se tratando de regras de limite superior, os processos terão suas requisições atendidas até que o limite superior seja alcançado. Assim que isso ocorrer, não será permitido que haja outra gravação ou leitura para os processos da regra até o início do próximo ciclo.

As requisições de processos que não fazem parte de nenhuma regra serão agendadas segundo o algoritmo padrão de escalonamento de disco. Contudo, serão satisfeitas depois que as requisições de processos participantes de regras de limite inferior sejam atendidas e antes dos processos afetados por regras de limite superior.

Espaço em Disco

O controle do espaço em disco é feito sem o uso de ciclos, visto que é um recurso com métricas absolutas. O controle aqui proposto não se destina à limitação de quanto um usuário pode alocar de espaço no sistema de arquivos, como é o caso do utilitário `quota` [Kar01]. O que se deseja aqui é impor limites, seja superior ou inferior, de tamanho dos arquivos abertos por um processo ou grupo de processos, que podem ser de um mesmo usuário, grupo de usuário ou de um grupo de processos correlatos, não necessariamente de um mesmo proprietário. Os valores dos limites das regras são expressos em tamanho físico real (KBs, MBs, GBs).

Esse tipo de controle pode ser utilizado quando se deseja garantir espaço em disco para uma determinada aplicação, como por exemplo um sistema gerenciador de banco de dados, que necessita primordialmente desse recurso para seu correto funcionamento. Também pode ser necessário impedir que usuários criem arquivos de tamanho elevado em pastas do sistema que não são geralmente gerenciadas por cotas, como por exemplo a pasta /tmp em sistemas Unix.

Para implementar esse tipo de controle é necessário saber antes da criação das regras o espaço disponível em disco. Isso porque não se pode reservar o recurso caso não haja espaço suficiente de armazenamento. Assim sendo, as regras de espaço em disco especificam tamanhos entre 0 e o espaço máximo disponível no momento da criação da regra.

Para as regras que especificam valores máximos de utilização do espaço é necessário somar o tamanho dos arquivos que estão sendo manipulados pelo processo (ou grupo), de forma a não permitir que mais dados sejam inseridos nos arquivos, nem tampouco outros

arquivos sejam abertos, caso essa soma ultrapasse o valor definido na regra. Por se tratar de um recurso não preemptível, não é possível fechar os arquivos ou deletar dados sem que haja comprometimento da continuidade de execução dos processos envolvidos. Logo, as únicas ações possíveis de serem tomadas caso o valor da regra seja alcançada são a restrição de abertura de novos arquivos e gravação de dados.

Em regras de limite inferior é feita a reserva do espaço em disco, que pode ser feita facilmente pelo sistema operacional sem a necessidade de grandes modificações no núcleo do sistema operacional. É lógico que o sistema operacional deverá manter a reserva atualizada à medida em que dados vão sendo gravados ou deletados pelos processos participantes da regra.

4.2.4 Rede

O gerenciamento da banda de rede funciona de maneira análoga ao controle de banda de disco. Aqui também é necessário saber qual a banda de rede máxima no momento da criação da regra para impedir que as regras especifiquem valores que ultrapassem a banda máxima.

Como outros recursos com métricas relativas, o controle de banda de rede requer a utilização de ciclos para estabelecer limites, que podem ser definidos por percentual de uso ou valores de banda (Kb/s, Mb/s, Gb/s).

O mecanismo para controle do uso da banda de rede consiste em:

1. enviar e entregar primeiramente os pacotes dos processos (ou grupos) controlados por regras de limite inferior, até que o limite seja ultrapassado;

2. enviar e entregar os pacotes dos processos não controlados por regras de acordo com a ordem de chegada (FIFO);

3. por último, enviar e entregar os pacotes dos processos (ou grupos) controlados por regras de limite superior enquanto o limite não for alcançado.

Deve-se notar que só se pode garantir a satisfação das regras de limite inferior localmente, já que não se pode interferir em outros sistemas que podem estar sobrecarregados ou mesmo não ter a banda requerida para a regra. Mais ainda, sendo o sistema dinâmico,

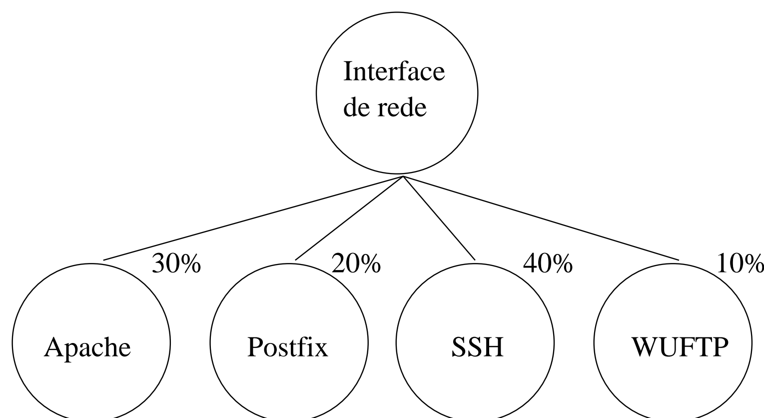

Figura 4.4: Compartilhamento da banda de rede através de regras.

não se pode pensar em um sistema de negociação no momento da criação da regra, visto que o processo pode ainda não ter feito suas conexões.

O modelo dinâmico para controle da banda externa que possibilite regras de limite inferior deve implementar um mecanismo de auto-negociação durante a criação das conexões[3] capaz de garantir o limite estabelecido pela regra ou, caso não consiga cumprir o limite, cancelar a conexão ou ainda informar ao administrador que o limite não foi atingido e qual o novo limite que está sendo usado.

Para que isso funcione pode-se utilizar o RSVP (Protocolo de Reserva de Recursos) em conjunção com WFQ (*Weight Fair Queueing*), que é um disciplinador para filas de pacotes, e assim construir uma arquitetura de serviços integrados capaz de garantir a banda de rede de um fim para outro em redes IP. O WFQ é o disciplinador padrão nos roteadores Cisco, e pode classificar o tráfego em diferentes fluxos baseado no endereço MAC, protocolo, portas de origem e destino, etc [Bal02].

As regras são definidas em termos de processos, ou seja, não são utilizadas as portas como método de classificação. Isso porque o modelo é implementado tendo processos como unidade básica. A figura 4.4 mostra a estrutura de compartilhamento de banda de rede entre os processos.

A implementação real desse controle pode ser baseado no CBQ (Class-Based Queueing), um modelo de gerência de pacotes de rede já disponível para Linux e outros sistemas operacionais [FJ95].

[3]Todos os sistemas envolvidos na conexão devem possuir o mesmo mecanismo de controle de banda para que seja possível a negociação, ou que seja utilizado um mecanismo padrão de alocação de banda de rede.

4.2.5 Recursos internos

Os descritores, os sockets e os semáforos são recursos internos do sistema operacional e podem ser controlados pela quantidade de cada um deles disponível para os processos.

Já a memória compartilhada, além desse controle, ainda permite estipular o tamanho da memória compartilhada destinada a um ou mais processos.

Descritores

Já foi visto que os descritores são utilizados pelos processos para obterem acesso aos arquivos, sockets, pipes, entre outras coisas. Para fazer isso o processo invoca a chamada de sistema `open()`. Essa chamada retorna um número inteiro que identifica de forma única o recurso acessado.

Usualmente os desenvolvedores de sistemas operacionais limitam o número de descritores que podem ser manipulados por um processo. No Linux, por padrão, um processo pode manter aberto no máximo 1024 descritores. Porém, esse valor pode ser facilmente alterado tanto para mais quanto para menos.

Para criar limites inferiores basta fazer a reserva do número requerido de descritores que serão usados para garantir este recurso aos processos da regra. Existe, no entanto, uma questão ao impor limites inferiores de descritores abertos por um processo (ou grupo) que deve ser levada em consideração: há um limite máximo de descritores abertos no sistema como um todo. Assim sendo, ao se criar uma regra de limite inferior deve-se verificar se este limite não vai ser atingido. Se for, a regra não deve ser criada.

As regras de limite superiores somente restringem quantos descritores podem ser abertos por um processo (ou grupo). O problema maior no controle dinâmico dos descritores está quando um processo já possui um determinado número x de descritores abertos e, de repente, o sistema recebe uma regra de limite superior informando que aquele processo (ou grupo) pode ter no máximo y descritores abertos, sendo y um valor inteiro positivo menor que x. Não se pode simplesmente negar o acesso aos descritores que já estão abertos, já que isso poderia acarretar problemas para a integridade daquele processo (ou grupo).

Para tratar desse problema, o sistema de controle dinâmico deve limitar o número de descritores abertos em x, não permitindo que novos descritores sejam abertos. Ao passo que os descritores vão sendo liberados, vai se decrementando o limite máximo de

descritores para aquele processo (ou grupo) até que o limite y seja atingido, resolvendo o problema e mantendo a integridade do processo.

Sockets e semáforos podem ser tratados de maneira análoga ao controle dos descritores. Assim, para implementação deste modelo para esses recursos pode-se utilizar o mesmo conceito aqui definido.

Memória compartilhada

A quantidade de áreas de memória compartilhadas que podem ser abertas por um processo (ou grupo) também pode ser tratada de maneira semelhante aos descritores. No entanto, a quantidade de espaço (*bytes*) que cada uma delas utiliza deve ser tradado por controle específico, pois este deve levar em consideração o tamanho de memória alocada pelo processo (ou grupo) para o compartilhamento.

As regras de limite inferior devem garantir que um processo (ou grupo) poderá alocar memória para a área compartilhada, reservando espaço em memória para tanto. Enquanto as regras de limite superior devem impedir que o processo (ou grupo) aloque mais espaço de memória compartilhada, em tamanho total, do que o definido pela regra.

4.3 Implementação do modelo

O modelo apresentado pode ser implementado modificando partes específicas do núcleo, como o escalonador para o controle do processador, o gerenciador de memória para o controle da utilização da memória física, da freqüência de faltas de página e uso da memória virtual, o classificador dos pacotes de rede para a gestão do uso de banda de rede, etc.

Essas modificações feitas no núcleo não afetariam o sistema de forma alguma até o momento que uma mensagem de criação de uma regra fosse recebida. A partir desse momento, os processos das regras estariam sob a política de controle dinâmico de recursos, obedecendo, então, aos critérios das regras.

As operações sobre regras podem ser feitas por chamadas de sistema que possuem as funções de:

1. criar regras:

`int setresourcelimits(recurso, escopo, identificacao, valor, tipo);`

2. remover regras:

   `int rmresourcelimits(recurso, escopo, identificacao);`

3. saber o estado da regra de um processo (ou grupo):

   `int getresourcelimits(recurso, escopo, identificacao, resposta).`

Os argumentos necessários para a chamada de sistema de criação de regras são: o recurso, o escopo, a identificação do escopo, o valor do limite e o tipo do limite a ser imposto.

O primeiro argumento informa ao sistema sobre qual controle deve ser aplicado ao recurso especificado. Pode-se assumir valores como utilização do processador, uso da memória física, freqüência das faltas de página, utilização da memória virtual, da banda de rede e de disco, do espaço em disco, quantidade de descritores, de semáforos, de sockets, de áreas de memória compartilhada e a utilização da memória compartilhada.

O parâmetro `escopo` notifica sobre qual granularidade a regra deverá agir. Possíveis valores são: processo, usuário, grupo de usuários, grupo de processos ou sessão.

Para identificar os processos atingidos pelo `escopo`, o argumento `identificação` recebe o *id* do alvo, como, por exemplo, o *pid* do processo, o *id* do usuário, o *gid* do grupo de usuários, etc.

Os últimos dois argumentos definem, respectivamente, o valor estabelecido para regra e o tipo do limite a ser imposto, inferior ou superior. O `valor` pode ser tanto percentual quanto absoluto, dependendo somente de `recurso`.

A chamada de sistema para remoção de uma regra deve possuir os seguintes parâmetros: o recurso, o escopo e a identificação do alvo. Esses parâmetros são utilizados para encontrar a regra dentro do modelo dinâmico.

Essa chamada deve primeiramente mover todos os processos alvo da regra em questão da política de controle dinâmico de recursos para a política de decisão comum do `recurso`, para depois apagar definitivamente a regra. Dessa forma, essa regra deixa de existir e os processos continuarão a sua execução de acordo com as políticas do sistema operacional utilizado.

A chamada de sistema de consulta a uma regra possui os mesmos argumentos da chamada de sistema de remoção de regra, que são utilizados para achar a regra no sistema,

Tabela 4.1: Diferenças entre os modelos alternativos de gestão de recursos e o modelo proposto.

	Limits	CKRM	Linux-SRT	DSRT	CPUCap	Este Modelo
Dinâmico	Não	Sim	Não	Não	Sim	Sim
Limite Superior	Sim	Sim	Sim	Não	Sim	Sim
Limite Inferior	Não	Sim	Sim	Sim	Não	Sim
Granularidade	Sessão	Classe	Classe	Aplicação	Processo	Processo Usuário Grupo de usuários Grupo de processos Sessão
Recursos	CPU Memória Internos	CPU Memória Disco Rede Outros	CPU Disco	CPU	CPU	CPU Memória Disco Internos Rede

mais a `resposta` a consulta, que é uma estrutura com os valores da regra, sobre qual recurso ela afeta, o escopo, o valor do limite que ela aplica, e se é uma regra de limite inferior ou superior. Um modelo de estrutura pode ser encontrado na seção 5.4.

Pode-se pensar também em uma chamada de sistema para alteração dos limites de uma regra. Ela deve receber como parâmetro, além do recurso, escopo e identificação, o novo limite a ser aplicado (inferior ou superior) e seu valor.

Com essas chamadas é possível manipular as regras do sistema para que o modelo seja utilizado segundo as necessidades do administrador do sistema ou da organização.

4.4 Conclusões

Este capítulo apresentou a proposta do modelo de gestão dinâmica de recursos, a finalidade do controle da maioria dos recursos computacionais e como tratar as excessões que ocorrem ao se impor limites de utilização dos recursos.

Pôde-se verificar a necessidade de um controle mais sofisticado nos recursos de maior importância para que o sistema possa ter o desempenho adequado às necessidades da organização ou do administrador do sistema.

As principais diferenças encontradas entre este modelo e as alternativas de controle de recursos apresentadas na seção 3.3 estão apresentadas na tabela 4.1.

Nota-se que o CKRM é o modelo alternativo de gestão de recursos que mais se apro-

xima desta proposta, no entanto a sua arquitetura visa tratar classes de processos criadas pelo administrador, enquanto este modelo define o controle dos recursos em estruturas pré-existentes em ambientes Unix. Ainda assim, o CKRM especifica um arcabouço para criação de controles para vários recursos diferentes, não os determinando, enquanto este modelo determina quais os recursos devem ser controlados e como isso deve ser feito.

No próximo capítulo será descrita em detalhes a implementação do controle de recursos para o processador e como são utilizadas essas funções para o seu gerenciamento dinâmico. Também será validada a eficácia do modelo apresentado e as conclusões obtidas quando o modelo está em execução.

Capítulo 5

Controle dinâmico de processador

Para validar as propostas que o modelo dinâmico oferece, foi implementado um protótipo cuja finalidade é aplicar este modelo na gestão de processadores no sistema operacional Linux e verificar se seu funcionamento é adequado e se sua utilização é viável em sistemas operacionais de mercado.

Este capítulo apresenta a construção desse protótipo, as mudanças realizadas no sistema, a avaliação dos resultados obtidos e custo computacional da implementação.

5.1 Descrição do problema

Na seção 2.2.1 foi visto como os sistemas operacionais de mercado gerenciam o uso do processador. Foi mostrado também que estes sistemas não fornecem uma forma de controle que permita aplicar regras de limites inferiores e superiores na utilização do processador de forma dinâmica.

Nota-se que com o mecanismo utilizado pelos sistemas atuais, um processo de baixa prioridade pode obter qualquer percentual de utilização do processador, incluindo a máxima, dependendo somente da disputa que há pelo uso da CPU. Não há meios de controlar isto, o que pode ser um problema para as organizações e também para os usuários que desejam que processos específicos tenham um limite superior de utilização do processador.

Também nota-se que com este mesmo mecanismo não se pode estipular que um ou mais processos tenham garantido um valor percentual mínimo de utilização da CPU. Isso porque a política de escalonamento dos processos é baseada nas prioridades, o que não

permite predizer a utilização da CPU pelos processos. Às vezes, no entanto, é necessário que alguns processos obtenham o processador por uma determinada quantidade de tempo para que ele possa ser adequadamente executado, como é o caso de processos com QoS. Uma das definições de sistema com QoS é que esse sistema deve ser capaz de interpretar, garantir e executar um serviço requisitado adequadamente [BB96], e isso só pode ser obtido por um modelo de controle que tolere e cumpra as especificações do software.

Outro problema que ocorre diz respeito ao impacto que cada processo sofre ao inserir um novo processo. A tabela 3.1 mostra o impacto na utilização do processador por processos de diferentes prioridades. Como pode-se perceber, todos os processos absorvem de forma semelhante o impacto no uso da CPU, diminuindo o tempo que cada processo tem de execução proporcional e aproximadamente igual. Isso pode não ser a decisão mais justa na divisão do uso do recurso, já que o processo de maior prioridade tem reduzida sua utilização do processador na mesma proporção que o processo de menor prioridade.

A seguir será apresentada a descrição da implementação do protótipo que utiliza o modelo de controle dinâmico para o gerenciamento do uso do processador visando resolver os problemas aqui descritos.

5.2 O escalonador de processos no Linux

Um algoritmo comumente utilizado para o escalonamento de processos é o de prioridades. Nesse algoritmo, os processos são classificados por um valor atribuído a eles de acordo com a sua relevância ou necessidade de uso do processador. Processos com alta prioridade são executados antes dos que possuem prioridade menor, enquanto processos de igual prioridade são escalonados de acordo com o algoritmo *Round Robin*. O Linux, em sua versão 2.4, da qual este trabalho trata, implementa esse método, sendo que quanto maior é a prioridade do processo, maior é o *quantum* que ele recebe. Enquanto o processo de mais alta prioridade possuir *quantum* para execução, ele mantém a posse do processador [Lov03].

Como outros sistemas operacionais, o Linux é um sistema multiprogramado, onde mais de um processo pode estar em execução ao mesmo tempo, no qual o *quantum* desses processos é definido como um múltiplo dos *ticks* do relógio de *hardware* [NFC+03]. Um contador denominado `jiffies` indica quantos *ticks* ocorreram no sistema desde sua

inicialização.

O escalonamento no Linux é do tipo preemptivo, ou seja, o processo é retirado de execução sem conhecimento ou interferência do próprio processo quando o seu *quantum* estiver esgotado[1].

O algoritmo de escalonamento do Linux divide o tempo de processamento em épocas (*epochs*). Uma época termina quando todos os processos capazes de serem executados tem seu *quantum* exaurido. Cada processo recebe um *quantum* calculado no início da época, que vai ser o tempo máximo de sua execução nesse período. Diferentes processos podem possuir diferentes valores de *quantum* [dOCT01]. Um processo pode ser selecionado para execução várias vezes em uma época, contanto que seu *quantum* não tenha acabado.

O núcleo do Linux implementa duas classes de prioridades: a primeira é aquela de valor *nice*, que é um número de -20 a 19, sendo que quanto menor o número mais alta é a prioridade, e antes o processo vai ser escolhido para execução. A fatia de tempo que um processo recebe para execução é calculada com base no valor *nice* do processo.

A segunda classe de prioridades é a utilizada exclusivamente por processos de tempo-real, e vão de 0 a 99. Esses processos só podem ser criados pelo super-usuário. Enquanto existirem processos de tempo-real a serem executados, os processos da primeira classe de prioridades não são selecionados para execução. A maioria dos sistemas Unix modernos implementam um esquema similar a este [Var96].

Com a finalidade de balancear o uso do processador, o escalonador monitora o uso da CPU pelos processos comuns. Se um processo ocupar o processador por muito tempo, ele tem sua prioridade diminuída para que processos com menor prioridade possam ser executados. Do mesmo modo, se um processo ficar muito tempo sem ser executado, a sua prioridade é aumentada. Essa característica do sistema Linux é chamada de prioridade dinâmica. O cáculo da prioridade dinâmica é feita como base no valor *nice* do processo, juntamente com o seu *quantum* restante.

Existem três políticas diferentes de escalonamento[dOCT01]:

A primeira política, que é chamada de `SCHED_FIFO` é utilizada somente por processos de tempo-real. Aqui um processo obtém o processador e não o deixa até que:

[1] O núcleo do Linux não é preemptivo, o que significa que um processo pode ser preemptado somente enquanto executando em *user mode* [BC00]. Processos do próprio núcleo ou processos de usuários que estão executando em *kernel mode* devido a uma chamada de sistema não são preemptíveis.

- um processo de tempo-real de prioridade superior torna-se apto a executar;
- o processo libere espontaneamente o processador para processos de prioridade igual à sua;
- o processo termine, ou seja bloqueado, em uma operação de entrada e saída ou de sincronização.

SCHED_RR é a segunda política, também é válida somente para processos de tempo-real e consiste na execução do processo até que uma das quatro situações ocorra:

- seu *quantum* tenha se esgotado;
- um processo de prioridade superior torna-se apto a executar;
- o processo libere espontaneamente o processador para processos de prioridade igual à sua;
- o processo termine ou seja bloqueado.

A política utilizada por processos comuns é a SCHED_OTHER. Consiste de um conjunto de filas de prioridades dinâmicas, sendo que os processos com maior prioridade são executados antes, e os que possuem prioridades iguais são escalonados um após o outro, repetidamente.

O escalonador do Linux é executado quando há uma chamada explícita à rotina do escalonador. Essa chamada ocorre quando o sistema:

- detecta que um processo deverá ser bloqueado em decorrência de uma operação de entrada e saída ou de sincronização;
- detecta que um processo esgotou seu *quantum* de execução;
- um processo de mais alta prioridade é desbloqueado pela ocorrência do evento que esperava;
- um processo explicitamente invoca o escalonador através de uma chamada de sistema do tipo *yield*[2].

[2]A chamada explícita à função de escalonamento feita a partir de um processo é conhecida como método *lazy* de execução.

5.3 O algoritmo de escalonamento proposto

O algoritmo proposto diferencia-se dos modelos comuns de escalonamento por permitir ao administrador do sistema quantificar o tempo de uso do processador pelos processos, e poder aplicar regras para o controle desse recurso dinamicamente.

Como já foi visto, uma regra é o meio de estabelecer quais são os recursos que devem ser dinamicamente controlados, sobre quais processos o controle incide e as condições de atuação. A regra pode ser aplicada diretamente sobre um processo ou uma *thread*, para os processos de um determinado usuário, para os processos de um grupo de usuários, para um grupo de processos, ou mesmo para uma sessão, conforme foi visto na figura 4.2. Esse método foi escolhido por utilizar estruturas nativas de ambientes Unix. Poderiam ter sido criadas novas formas de agrupar processos, tal como foi feito por [NFC+03], porém, para aproveitar os recursos já disponíveis no sistema, preferiu-se manter o padrão Unix.

Seguindo o que foi definido no modelo, para que seja possível notificar o núcleo da existência de uma nova regra, foi criada uma nova chamada de sistema, denominada `resourcelimits`, que possui os seguintes parâmetros:

- `resource`: identifica qual o recurso a ser controlado. No protótipo implementado somente o valor `CPUPERC` (percentual de uso do processador) está disponível.
- `scope`: define quem será afetado pela regra, ou seja, o seu escopo. Pode assumir os valores: `RLPROCESS`, `RLPROCESSGROUP`, `RLSESSION`, `RLUSER` e `RLGROUP`, correspondendo respectivamente ao controle de um processo, de um grupo de processos, de uma sessão e dos processos de um usuário ou de um grupo de usuários.
- `target`: informa o identificador do escopo em questão, ou seja, o *id* do processo, do usuário, do grupo de usuários, do grupo de processos ou da sessão.
- `value`: especifica o valor numérico do limite a ser imposto.
- `type`: define o tipo do limite, que pode ser `UPPER`, para limite superior, e `LOWER`, para limite inferior.

O valor de retorno da chamada de sistema depende do sucesso ou do erro ocorrido na execução da chamada de sistema. O valor 0 (zero) representa êxito, enquanto valores negativos informam que um erro ocorreu. Entre os erros possíveis estão a inexistência do

identificador definido por `target` e a impossibilidade de definir um limite inferior, caso a soma dos limites já definidos com o novo limite ultrapasse 100% de uso do recurso.

Assim que a chamada de sistema `resourcelimits` é executada, o sistema reage sobre os processos pertinentes, colocando-os sob uma nova política chamada `SCHED_RL` (escalonamento via `ResourceLimits`) e controlando os recursos para que sejam respeitadas as condições da regra. A condição se refere ao uso máximo permitido daquele recurso, ou ao mínimo reservado para aquele recurso. Nesse caso, é um percentual relativo a um período de tempo pré-determinado, `CPUPERC`.

`SCHED_RL` segue a política definida pelo modelo, que pode ser encontrada na seção 4.2.1.

Assim como proposto, o controle de acesso é feito durante a inserção ou alteração da regra. Uma regra que restrinja o uso do recurso pode ser feita pelo usuário proprietário dos processos em questão. Já as regras que privilegiam o uso do recurso podem ser aplicadas somente pelo administrador do sistema. Regras de limite máximo que reflitam sobre processos de mais de um usuário também só podem ser aplicadas pelo super-usuário.

Como poderá ser visto no apêndice A, no momento da definição de regras de limite inferior é feita a verificação para garantir que a soma de todas as regras de limite superior não ultrapasse 100% de utilização de CPU, para evitar que as regras excedam a capacidade física máxima de execução dos processos.

Na seção 5.4 poderá ser visto que a reserva de recursos está implícita no algoritmo desenvolvido, pois quando houver uma regra de limite inferior, os processos pertinentes a ela são executados antes de todos os outros, garantindo o cumprimento da regra. No entanto, esse método acarreta atraso na execução de outros processos que não fazem parte de regras de limite inferior.

A próxima seção entrará mais em detalhes da implementação do mecanismo de controle dinâmico do processador.

5.4 Descrição da implementação

O protótipo de controle dinâmico do processador foi implementado no Conectiva 9.0 sobre núcleo do Linux versão 2.4.21. A maior parte das alterações feitas foram aplicadas sobre o próprio escalonador do sistema. A lista de arquivos do núcleo alterados para a

criação do protótipo e suas modificações é a seguinte:

- `entry.S`: definição da chamada de sistema `resourcelimits`;
- `init_task.c`: inclusão da inicialização das listas e variáveis utilizadas pelo protótipo;
- `unistd.h`: definição do número da chamada de sistema `resourcelimits`;
- `sched.h`: criação da política `SCHED_RL` e inclusão de um novo campo na estrutura dos processos;
- `exit.c`: deleção de regras na finalização dos processos, se necessário;
- `fork.c`: inclusão de novos processos em suas respectivas regras;
- `sched.c`: implementação da política `SCHED_RL`;
- `sys.c`: implementação da chamada de sistema `resourcelimits`;
- `timer.c`: verificação do cumprimento das regras a cada *tick* do relógio de *hardware*.

Também foi incluído o arquivo `resourcelimits.h` que possui definições de estruturas utilizadas em outras partes do sistema.

No protótipo apresentado foi definida uma nova estrutura para armazenar as informações específicas para regras de controle do uso do processador. Essa estrutura é chamada de `TRLGroup`. Com essa estrutura, foi criada uma lista que em cada nó guarda, além dos parâmetros de uma regra, mais um campo para armazenar informações pertinentes ao uso do processador e outro campo para apontar para o próximo nó de regras. Cada nó é chamado de *grupo*.

```
struct TRLGroup {
        int resource;
        int scope;
        int target;
        long value;
        int type;
        int RLUSEDTICKS;
        int active;
        int number_of_elements;
```

```
        struct list_head list;
};
```

Houve também a inserção de uma nova variável no Bloco de Controle de Processos (PCB) para que cada processo aponte para o seu *grupo*, caso pertença a um deles.

Uma variável global, `RLNEXT`, armazena a informação de quando se iniciará o próximo ciclo do sistema. Um ciclo do sistema é definido por um determinado número de vezes que ocorre uma interrupção de tempo, sendo que no Linux a variável `HZ` (freqüência do *timer*) armazena esse valor[3]. O tempo de duração do ciclo é de 1 (um) segundo[4]. A cada ciclo do sistema a regra deve manter a utilização do processador dentro do limite especificado. Quando `jiffies` for maior que `RLNEXT`, o valor de `RLNEXT` é atualizado de acordo com `RLNEXT = jiffies + HZ`. Nesse mesmo momento o escalonador é chamado para recalcular os valores de utilização do processador por cada *grupo*.

Esse cálculo consiste em determinar quantos *ticks* de relógio essa regra possui nesse ciclo. A variável da regra que armazena esse valor é `RLUSEDTICKS`, e é um valor normalizado de acordo com `HZ` já que este pode assumir diversos valores dependendo da arquitetura ou da própria distribuição.

A cada *tick* do relógio a variável `RLUSEDTICKS` é decrementada. Se a regra especifica um limite máximo, quando `RLUSEDTICKS` atinge zero os processos do escopo da regra são impedidos de executar até o próximo ciclo. Isso é feito na função `update_one_process()` do controle de tempo do sistema, e pode ser visto na listagem de código abaixo.

```
void update_one_process(struct task_struct *p, unsigned long user,
                        unsigned long system, int cpu)
{
/* resourcelimit */
if (RLNEXT < jiffies){
        RLNEXT = jiffies + HZ;
        need_recalculate = 1;
        if (there_are_lower_rules)
                p->counter = 0;
}
```

[3]No Linux 2.4, HZ é igual 100 para arquitetura x86.

[4]O tempo de 1 (um) segundo foi utilizado por ser o valor pelo qual o *timer* é ajustado. Ao se ajustar, por exemplo, HZ para 100, o *timer* irá gerar 100 interrupções de relógio no sistema durante um segundo, ou seja, de 10 em 10ms. Isso define a granularidade do sistema: *timers* ajustados em intervalos de 10ms, *quantum* medido em intervalos de 10ms, etc [Lov02].

```
if (p->policy == SCHED_RL){
        if (--p->rlgroup->RLUSEDTICKS == -1) {
                struct task_struct *pp;
                p->rlgroup->RLUSEDTICKS=0;
                for_each_task(pp){
                        if (pp->rlgroup == p->rlgroup)
                                pp->counter = 0;
                }
        }
}
p->per_cpu_utime[cpu] += user;
p->per_cpu_stime[cpu] += system;
do_process_times(p, user, system);
        ...
}
```

A função `goodness()` do núcleo do Linux é responsável por atribuir peso aos processos para que eles possam ser escolhidos pelo escalonador. Esse peso é baseado em quanto tempo de execução o processo ainda possui. Essa função foi alterada de forma que pudesse fornecer pesos muitos baixos caso o processo atingisse seu limite superior. No Linux, um processo não é executado se o seu peso for igual a -1000. No caso de regras de limite inferior, o processo terá o seu peso aumentado para que sempre seja escolhido para execução enquanto o limite requerido não for atingido. A listagem abaixo mostra a parte da função `goodness()` que é responsável pela atribuição dos pesos aos processos. Os processos que sofrem essa alteração de peso são aqueles que fazem parte da política `SCHED_RL`.

```
if (p->policy == SCHED_RL){
        /* resourcelimits */
        weight = p->counter;
        if (!need_recalculate){
                if ( p->rlgroup->type == L_UPPER){
                        if (! p->rlgroup->RLUSEDTICKS)
                                weight=-1001;
                }
                else
                        if (p->rlgroup->RLUSEDTICKS){
                                weight += (KEEP_RUNNING + p->counter);
```

```
                    }
                    else{
                            p->counter = NICE_TO_TICKS(20); //menor
                                prioridade
                            weight = p->counter;
                    }
            }
            goto out;
    }
if (p->policy & SCHED_YIELD)
        goto out;
```

Na inicialização dos processos, foi necessário avaliar se os processos que estão sendo criados estão associados a alguma regra. Se isso ocorrer, os processos em questão são adicionados ao *grupo* pertinente e são executados nos limites impostos pela regra. Essa verificação é feita na função `do_fork()`.

A figura 5.1 mostra como é o funcionamento do sistema de controle dinâmico de recursos e os principais locais de alterações feitas no núcleo do Linux. A ordem de execução do controle dinâmico é mostrado na figura pela numeração nos objetos do fluxograma. A lista abaixo explica cada um dos passos de acordo com sua referência numérica:

1. Ao ocorrer um *tick* do relógio é disparada a verificação dos limites do processo em execução se ele pertencer a uma regra;

2. Se o ciclo se completou, o escalonador é chamado para escolher um novo processo e os tempos de uso do processador são recalculados;

3. Caso o ciclo ainda não tenha terminado, o grupo do processo em execução tem o seu tempo de uso do processador atualizado;

4. Se a regra for de limite superior e o limite foi atingido, o processo é tirado de execução, senão continua sua execução se o seu *quantum* não for zero;

5. Se a regra for de limite inferior e o limite não foi atingido, um processo do grupo recebe o processador para continuar sua execução, senão continua a execução concorrendo pelo processador junto com os processos comuns;

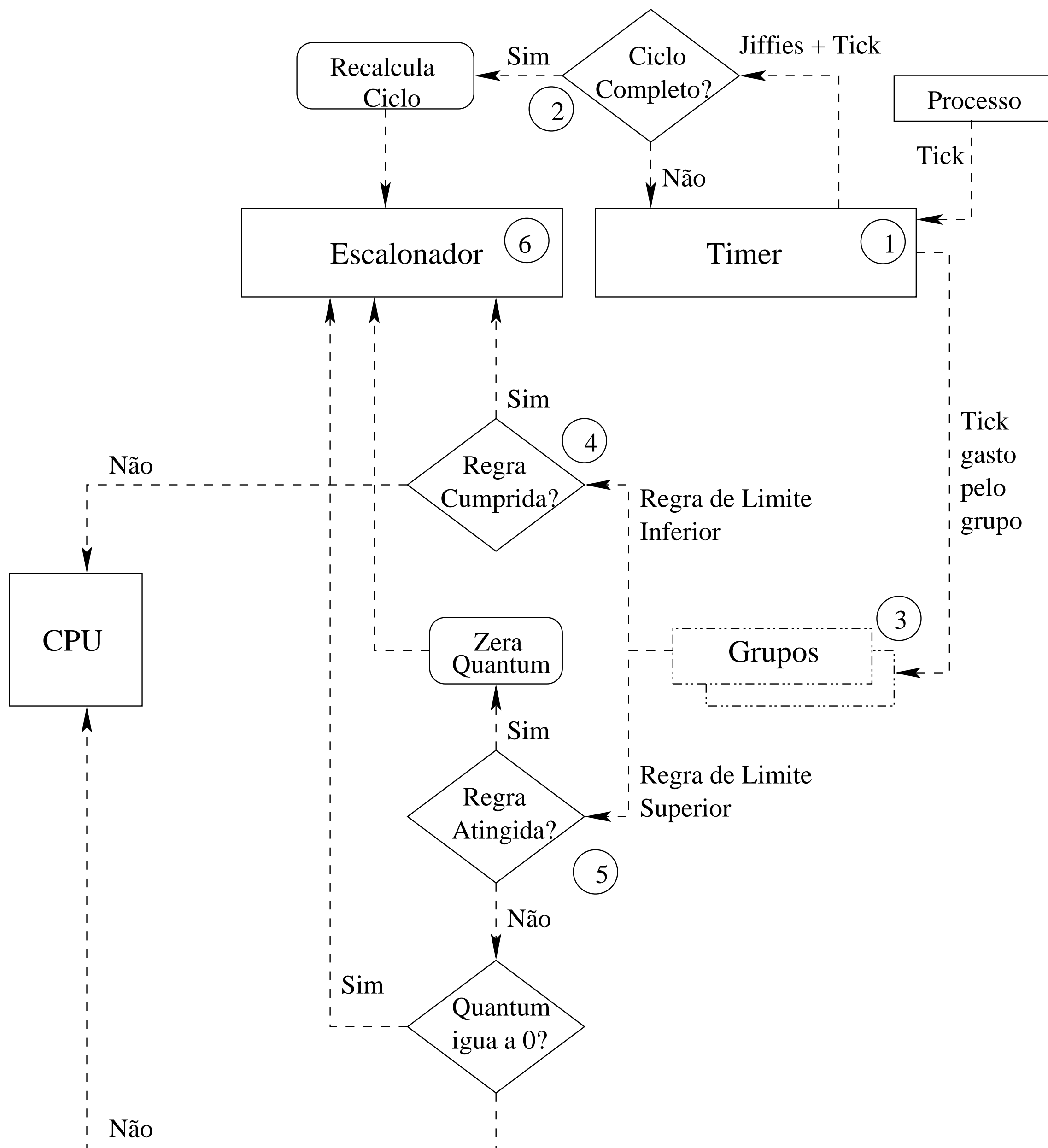

Figura 5.1: Fluxograma do controle dinâmico de recursos.

6. O escalonador escolhe o processo mais apropriado para execução seguindo as definições das regras.

Com essa implementação, os processos de regras de limite inferior são executados antes de todos os outros processos no início do ciclo, sendo classificados para execução de acordo com sua prioridade, seguido dos processos pertencentes à regras de limites inferiores, superiores e processos normais, sendo que a partir desse momento eles disputam a CPU de acordo com suas prioridades, no entanto, os processos de regras de limite superior só são executados até o momento que atingem seu limite.

Na próxima seção serão vistos os testes e resultados obtidos com esse protótipo.

5.5 Avaliação do protótipo

O protótipo foi testado em um servidor Athlon XP 1700+, com 256 MBytes de memória, rodando a distribuição Slackware 9 com o núcleo 2.4.21. Foi criado um pequeno programa com um laço infinito, chamado *usecpu* para que se pudesse avaliar eficácia do sistema. Esse programa faz uso intenso da CPU, de forma que se não estiver sendo executado nenhum outro processo ele utilizará 100% do processador.

As informações sobre uso do processador foram obtidas pelo utilitário `top`, com taxa de atualização de 1 (um) segundo e por um monitor de utilização da CPU implementado especificamente para este projeto que permite selecionar intervalos de tempos em escala de nanosegundos[5]. O código fonte desse monitor encontra-se em anexo, no apêndice B.

Um pequeno programa foi escrito para que se pudesse iniciar e alterar as regras. Ele possui os mesmos parâmetros da chamada de sistema `resourcelimits` e somente repassa as informações recebidas para a chamada. Esse programa também pode ser encontrado no apêndice C.

Vários experimentos foram executados para validar a proposta. O primeiro experimento consistiu em executar vários processos *usecpu*, sendo aplicado um limite superior de uso da CPU a um deles. Na figura 5.2 o processo P1 está sendo limitado a 10% da CPU por esta regra. Mostra-se o comportamento temporal deste processo quando disputando

[5] A fidelidade a essa escala depende exclusivamente da implementação da função *nanosleep*. Como ela é baseada no mecanismo normal de *timers*, sua resolução é de $1/HZ$, ou no núcleo 2.4 do Linux, 10ms [Pro99b].

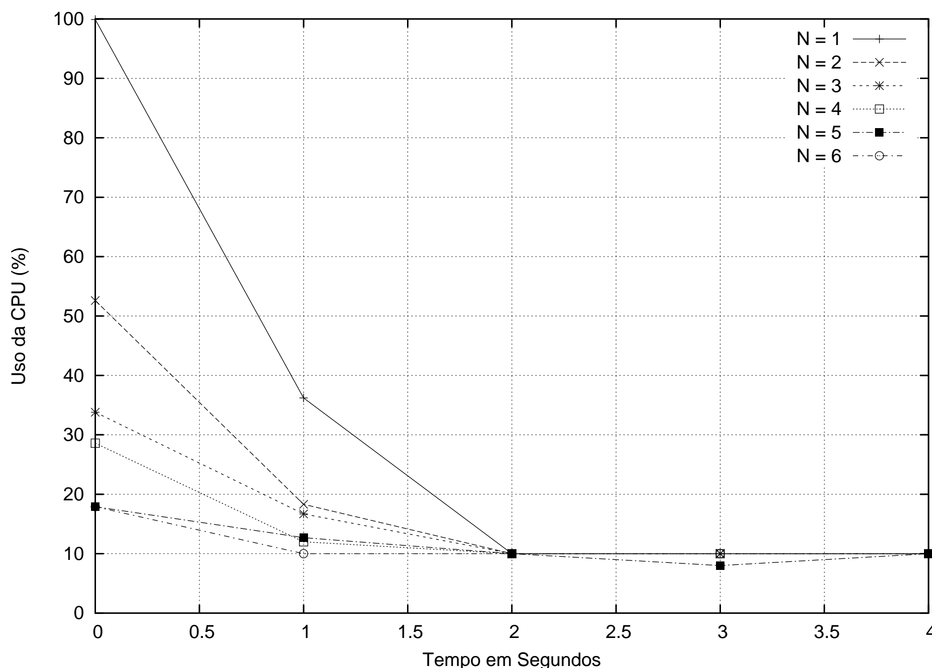

Figura 5.2: Uso da CPU por P1, com N processos disputando a CPU.

o uso do processador com N processos, e N variando de 1 (somente P1) a 6 (P1 e mais 5 processos). A regra de definição do limite é aplicada sobre P1 em t=0.

Pode-se observar que o sistema leva cerca de 2 segundos para estabilizar o processo no limite estabelecido. Esse período de transição decorre do algoritmo de escalonamento do Linux, do valor do ciclo escolhido e da própria granularidade das medidas efetuadas (1 segundo).

O mesmo experimento foi repetido com grupos de processos, sessões e usuários, com resultados bastante similares. Em nenhuma das execuções realizadas algum processo com limite máximo imposto excedeu o especificado pela regra. A figura 5.3 apresenta P1 e P2 em execução, sendo que eles pertencem a um usuário que tem a utilização do processador limitada para 5% no máximo. Pode-se observar que a soma da utilização do processador por esses processos não foi superior ao estipulado pela regra.

Para os testes com limites mínimos de utilização do processador foi empregado o mesmo processo *usecpu*. Fixando o limite inferior de uso da CPU para P1 em 90% e variando o número total de processos disputando o processador (de 1 a 6), foram obtidas as curvas apresentadas na figura 5.4.

Foram também realizados testes de limite mínimo para outros escopos. Na figura 5.5 podem ser vistos os processos P1 e P2 que fazem parte de uma mesma sessão, que possui

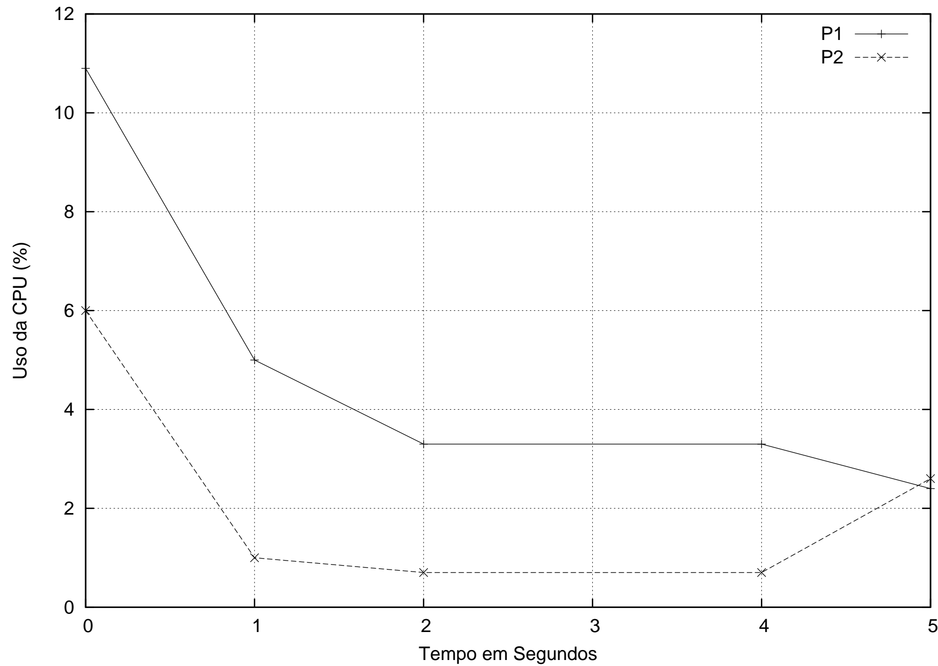

Figura 5.3: Uso da CPU por P1 e P2 pertencentes a uma regra limite superior.

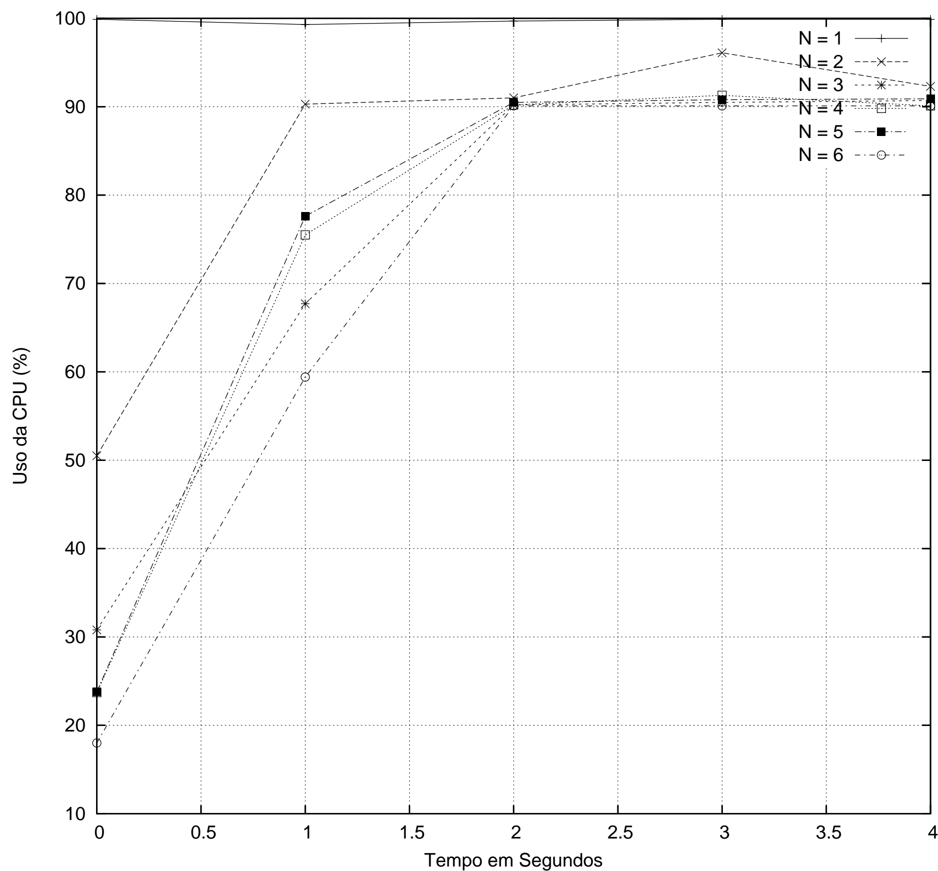

Figura 5.4: Uso da CPU por P1, com N processos disputando a CPU.

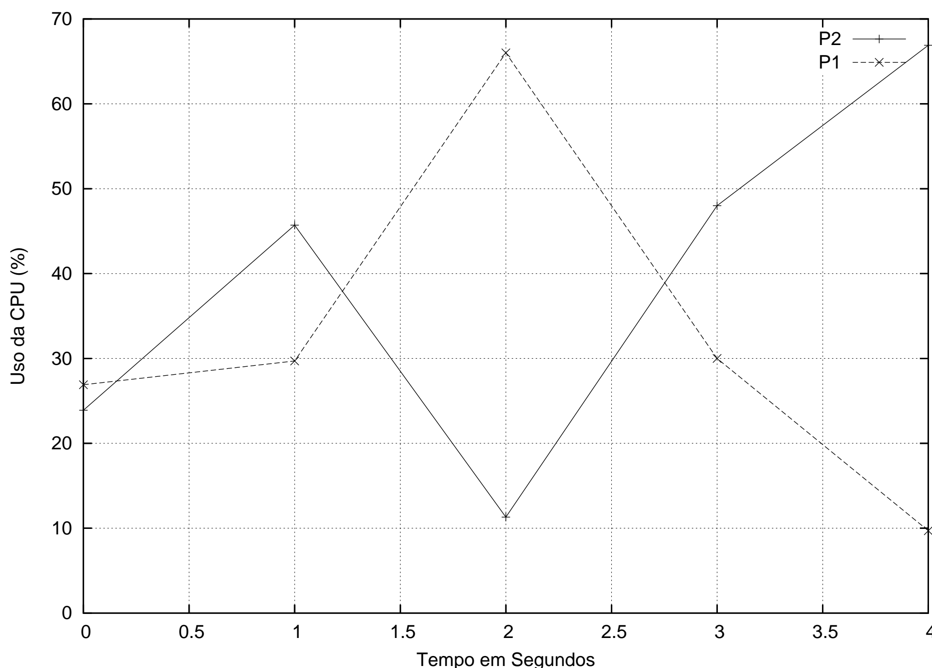

Figura 5.5: Uso da CPU por P1 e P2 pertencentes a uma regra limite inferior.

uma regra que define o uso mínimo do processador para 60%.

Pode-se notar que o sistema não faz o balanceamento do uso do processador entre os processos internos do *grupo* durante o período de um ciclo. Isso ocorre porque ao se iniciar a execução de um processo com limite mínimo, ele mantém a posse do processador até atingir o limite. Porém, no próximo ciclo ele terá peso diminuído em relação aos outros integrantes do ciclo, tornando a execução balanceada entre os processos do *grupo* no período total de execução.

Devido a arquitetura de *grupos* empregada no modelo, a dinâmica de adaptação do sistema foi similar para todos tipos de regras, sejam de limite inferior ou superior sobre qualquer escopo.

Como pode ser visto, em nenhum momento o algoritmo deixou de cumprir os limites definidos nas regras. Também é possível comprovar que o modelo implementa um mecanismo dinâmico, visto que as alterações nas regras são aplicadas rapidamente, entre 1 e 2 segundos, sem que seja necessário reiniciar os processos. Nota-se também que com esse modelo o impacto de uso da CPU durante a inserção de um novo processo será distribuído somente entre processos que não possuam regras de limite inferior, já que os processos que a possuem respeitarão os limites das regras.

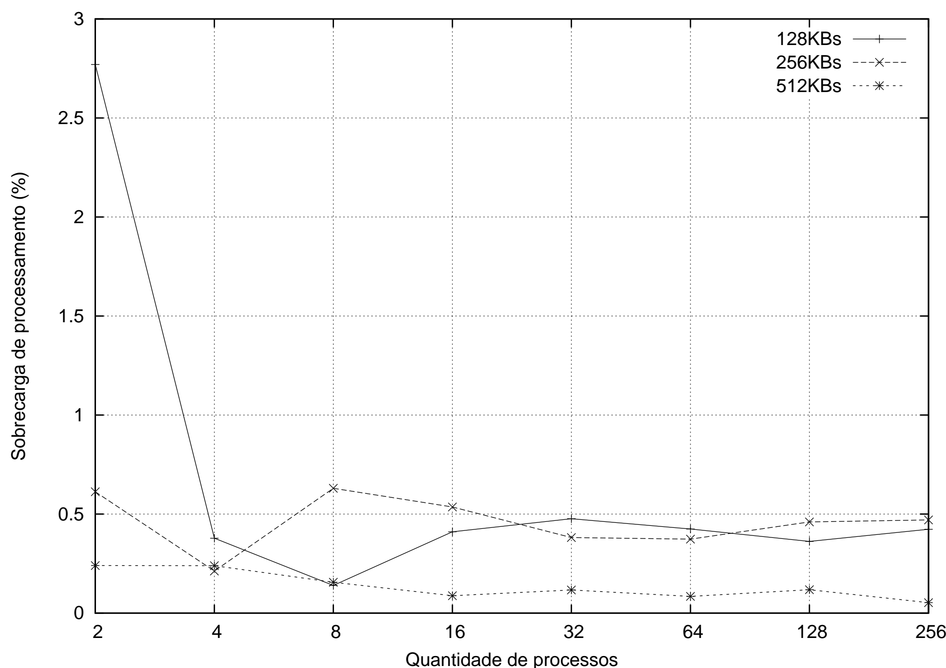

Figura 5.6: Acréscimo de tempo de execução imposto pelo mecanismo de aplicação das regras.

5.5.1 O desempenho e custo da implementação

Para quantificar a sobrecarga de processamento no núcleo do Linux foi utilizada a ferramenta de análise de desempenho `lmbench` [MS96]. Esta ferramenta compara os desempenhos para ambientes Unix e é capaz de examinar a latência da troca de contexto, da criação de processos, da manipulação de sinais, da leitura de memória e de muitos outros parâmetros.

O *benchmark* foi realizado primeiramente no mesmo sistema com o núcleo inalterado (kernel 2.4.21). Logo em seguida, o mesmo experimento foi feito sobre o núcleo com o modelo implementado. Os parâmetros analisados foram o número de processos disputando a CPU (de 2 a 256) e tamanho dos processos em memória (de 128 Kbytes a 512 Kbytes). O gráfico da figura 5.6 apresenta os resultados obtidos. Nele pode-se observar que o custo da implementação desse modelo quando não há regras criadas é normalmente abaixo de 1% do tempo total de processamento.

A ferramenta `lmbench` não apresentou nenhuma outra alteração no desempenho do sistema durante a realização dos testes, tampouco foi sentido qualquer decréscimo na velocidade de execução dos aplicativos.

5.6 Conclusões

Neste capítulo foi apresentada a implementação de um protótipo com o modelo de controle dinâmico do uso do processador. Ele permite que o administrador ou usuários dos sistemas possam quantificar o uso do processador por um processo ou vários processos. Conforme observado, o protótipo confirma o que o modelo propunha, tornando-o viável. Com ele, aplicativos que necessitem de controle dinâmico de recursos, como por exemplo aplicativos com QoS, podem ser mais facilmente implementados, bem como o sistema pode ser customizado para um melhor atendimento dos requisitos da organização que o utiliza. Também é possível através dele implementar um mecanismo de respostas a ataques de negação de serviço [dAMF03], restringindo o uso dos recursos por processos que estejam comprometidos, mas mantendo-os ativos o tempo necessário para uma auditoria de segurança ou análise forence.

Do ponto de vista funcional, o protótipo segue o previsto no modelo. Em termos práticos várias melhorias podem ser feitas, como, por exemplo, o balanceamento da utilização da CPU pelos processos que pertencem a um *grupo* com limite inferior durante o ciclo. Também é possível fazer algumas otimizações no código para que a sobrecarga seja ainda menor.

O desempenho do protótipo foi satisfatório tanto em relação a velocidade quanto aos resultados. Ele não afetou a integridade dos processos que não são gerenciados por regras, mesmo quando diminuindo o acesso que possuíam aos recursos em prol dos que possuíam regras de uso mínimo do processador, ao mesmo tempo que preservou os processos sob gerência de regras.

A disponibilidade e a estabilidade do sistema não foram em nada comprometidas, visto que uma nova política de execução foi criada, mantendo inalterada as políticas originais que ainda são usadas pelo processos comuns do sistema. A política `SCHED_RL` também mostrou-se estável, não prejudicando o sistema em qualquer forma.

Nesse ponto, verifica-se que é necessária a implementação do controle de outros recursos, tais quais memória, descritores, banda de disco, banda de rede e etc., para que o modelo esteja completamente funcional e assim poder estar disponível em sistemas operacionais de mercado.

Capítulo 6

Considerações finais

6.1 Resumo do trabalho realizado

Este trabalho visou apresentar um modelo de controle dinâmico de recursos do sistema operacional. Ele tem por objetivo permitir que os usuários do sistema quantifiquem o uso dos recursos pelos processos, aplicando condições de utilização. O modelo define limites inferiores e superiores para uso dos recursos, tornando possível a garantia ou a restrição de uso dos recursos, respectivamente.

Este modelo permite a co-existência de outros métodos de escalonamento no mesmo sistema operacional, mantendo as características do sistema operacional hospedeiro.

O modelo também permite agrupar os processos de diversas maneiras para que regras de limites inferiores e superiores possam ser aplicadas a grupos de processos relacionados, conforme a necessidade da organização. Isso tudo é feito de forma dinâmica, ou seja, sem que haja necessidade de reiniciar o processo ou a sessão do usuário para que as condições sejam aplicadas. As regras começam a atuar assim que são criadas, tomando somente um pequeno período de adequação do sistema até que as condições possam ser atingidas.

A viabilidade do modelo foi verificada pelo protótipo aqui apresentado, onde o uso do processador é controlado conforme o modelo proposto. Os testes realizados mostraram que o comportamento do sistema como um todo correspondeu ao esperado, garantindo e restringindo a utilização do processador pelos processos.

Pode-se afirmar que o custo de implementação do modelo é baixo, e que o impacto no desempenho do sistema é modesto, tornando o modelo viável de implementação e, muito

possivelmente, de aplicação em sistemas operacionais de rede de mercado.

6.2 Principais contribuições

Entre as principais contribuições do modelo, a quantificação explícita do uso dos recursos é a que mais se destaca. Com isso uma melhor noção de compartilhamento dos recursos é obtida, provendo um mecanismo de controle de recursos de alto nível mais adaptativo às necessidades da organização.

O modelo permite que programas que desejam definir expressamente o uso dos recursos do sistema possam ser desenvolvidos sem que sejam necessárias alterações no núcleo do sistema, ou mesmo conhecimento específico sobre o controle interno dos recursos.

Isso é válido, por exemplo, para programas que exigem limites superiores na utilização dos recursos, como um sistema de respostas a ataques, limitando a execução de processos que estejam comprometidos e com isso preservando o valor forense das informações do sistema [dAMF03].

E finalmente, o sistema permite que haja um melhor dimensionamento no uso do sistema, inclusive no modo do núcleo, o que não ocorre nos sistemas operacionais modernos. Assim, o administrador pode ter mais clareza na decisão de onde alocar suas aplicações e qual o uso que elas farão dos recursos computacionais do sistema em que estão hospedadas.

Outra contribuição deste trabalho foi a publicação dos resultados preliminares desta proposta em [SMJ04].

6.3 Limitações

Uma das principais limitações do sistema consiste em não poder agrupar processos que de alguma forma estão relacionados, mas não possuem nenhum escopo fornecido pelo modelo em comum. Dessa forma, para que esses processos relacionados possam ser juntamente controlados é necessária a criação de mais de uma regra, não permitindo que esses processos compartilhem o uso dos recursos. Por exemplo, seja o processo P1 e P2 pertencentes a uma aplicação relevante para a organização. O administrador deseja que o processo P1 e P2 juntos possuam um limite superior de 60% para o uso do processador. Porém, P1 e P2 não possuem nenhum escopo em comum (usuário, grupo de usuário,

grupo de processo ou sessão). É, então, criada duas regras, uma para o processo P1 e outra para o processo P2, que especificam que o limite superior de cada um deles é de 30%, o que significa que o uso total do processador seria de no máximo 60%. Mas como os processos são controlados por regras distintas, eles não compartilham o uso do recurso, assim, se o processo P1 for bloqueado, P2 utilizará no máximo 30% do processador, sendo que se participassem da mesma regra, P2 poderia utilizar os 60% destinados àquela regra. A única solução para esse problema é fazer os processos possuírem um escopo em comum, o que nem sempre é possível.

Outra limitação que ocorre no protótipo é que um processo pode fazer parte apenas de uma regra do uso de cada recurso. Assim, limitar o uso dos recursos pelo processo entre um intervalo mínimo e máximo não é possível. Por exemplo, se se deseja ter P1 utilizando o processador entre 30% e 50%, poderia-se criar uma regra de limite superior a 50% e outra de limite inferior a 30%, porém no estágio atual de desenvolvimento do protótipo isso não é possível, já que cada processo só pode ser afetado por uma regra.

Isso pode levar ao aperfeiçoamento do modelo para permitir que sejam criadas regras de intervalo, que seriam utilizadas para a definição de limites superiores e inferiores ao mesmo tempo em uma única regra, a fim de resolver esse problema.

Outra limitação são os recursos que são compartilhados por mais de um processo e que pertencem à regras distintas para o controle do recurso compartilhado. Esses recursos podem ser memória compartilhada, arquivos de mapeamento de memória e memória de *thread*. Impor limites a esse tipo de recursos pode ser complicado. Se, por exemplo, ao ajustar o limite superior de memória compartilhada de um processo para 4 páginas, e outro processo que utiliza o mesmo compartilhamento de memória possui uma regra de uso de memória compartilhada com limite superior definida em 6 páginas, pode fazer com que uma das regras não seja satisfeita, visto que a memória compartilhada pode atingir até 6 páginas, não cumprindo a regra que diz que deveria usar no máximo 4. Esse problema pode ser resolvido definindo que será adotado o menor limite entre as regras dos processos envolvidos no compartilhamento dos recursos.

6.4 Trabalhos correlatos

Vários esforços vêm sendo realizados para um melhor controle dos recursos do sistema operacional, porém, essas estruturas precisam ser incorporadas dentro do sistema operacional [DS96]. A maioria das pesquisas sobre controle de recursos estão ligadas à qualidade de serviço, já que esta necessita daquela.

Até o momento foram encontrados alguns trabalhos correlatos a este que foram apresentados no capítulo 3, na seção 3.3, que são o CKRM, o Linux-SRT e o DSRT. Na mesma linha de pesquisa, pode-se citar ainda:

- o *escalonador hierárquico de CPU para sistemas operacionais multimídia* [GV96]: a banda de CPU é particionada em várias classes de aplicações, onde os processos são alocados. Implementado para o Solaris, ele é capaz de dinamicamente garantir a alocação de uso do processador. Uma implementação desse modelo também está disponível para Linux em [Dei05].
- o *QLinux* [SCG+00]: também dividindo a banda de CPU em classes, o QLinux pode prover uma taxa garantida de uso do processador, com ajustes dinâmicos. O mesmo pode ser feito para a banda de disco.

Esses dois trabalhos foram citados aqui, e não na seção 3.3, devido as suas semelhanças com o Linux-SRT, tornando redundante naquele momento a sua descrição.

Outros trabalhos aqui descritos são de esforços em sistemas com QoS, pois para poder garantir a execução adequada de um serviço, o sistema operacional deve fornecer mecanismos de controle dinâmico de recursos.

Um controle mais sofisticado dos recursos de rede foi proposto por [GR02]. O proposto é que o gerenciamento dos pacotes de rede deixe de ser feita por uma *thread* que utiliza o tempo do processador reservado para a execução do núcleo, e passe a ser executado no tempo destinado ao processo que requisitou aqueles pacotes de rede. Com a velocidade de placas de redes cada vez mais rápidas, por muitas vezes os processadores não conseguem receber todos os pacotes por não poderem processar todos eles. Retirando o controle da *thread* do núcleo, menos computação é necessária, já que o processo que recebe os pacotes de rede está utilizando o processador no momento da chegada, não sendo necessário fazer uma interrupção de software para os receber.

[BB96] sugere um modelo de QoS genérico em um sistema operacional e implementa a proposta sobre o SunOS, utilizando para tanto uma classe de programação em tempo-real. É sugerido um modelo de gerenciamento de recursos que pode trabalhar com reserva de recursos e também com preempção de recursos. No primeiro caso, antes de ser confirmada a execução do serviço é feita a reserva dos recursos necessários, caso não seja possível executar o processo nas condições mínimas requisitadas, o processo não é executado. Já no segundo caso, se um processo deseja utilizar os recursos do sistema e o mínimo requisitado não está disponível, processos com prioridade menor terão seus recursos restringidos para que ele possa ser executado.

No gerenciamento de aplicações distribuídas, [NBB97] implementa um protótipo que consegue garantir qualidade de serviço para aplicações multimídia em um sistema operacional distribuído. Desenvolvido sobre Chorus/Mix, que é um sistema modular baseado em Unix, o protótipo gerencia os recursos do computador, tais como processador, rede e disco, a partir de componentes individuais, para poder prover QoS ao sistema operacional.

[YC01] apresenta o modelo e a implementação de gerência de recursos em sistemas operacionais de roteadores. É definida uma abstração da alocação de recursos, que pode escalonar vários recursos de tempo e espaço compartilhado com garantia e diferenciação de qualidade de serviço. Um classificador flexível e escalável de pacotes habilita a alocação dinâmica de recursos durante o processamento dos pacotes recebidos. Foram integrados também o controle de tempo de processamento, banda de *forwarding*, capacidade de armazenamento em memória e em dispositivos de armazenamento secundário.

Para garantir qualidade de serviços em sistemas de arquivos, [Bar97] propõe um mecanismo para controlar o acesso ao disco. Controles de banda, proteção e acesso compartilhado ao sistema de arquivos são fornecidos por um módulo de controle. Esse módulo recebe as informações do módulo de dados que é responsável por prover ao usuário um método de acesso ao sistema de arquivos. Com esse método, foi possível dar suporte à qualidade de serviço no sistema de arquivos.

Muitos outros trabalhos estão relacionados a suporte de QoS em sistemas operacionais, mas não foi encontrado nenhum outro que, especificamente, controle os recursos do sistema permitindo aplicação de limites superiores e inferiores.

6.5 Perspectivas e trabalhos futuros

As perspectivas deste trabalho são relativas à extensão do protótipo no sentido de implementar o controle de outros recursos, como quantidade de memória física (RAM), banda de disco e de rede.

Também há necessidade da implementação do controle dinâmico do processador no novo escalonador O(1) [Mol05] do núcleo do Linux 2.6. O novo escalonador é baseado em filas múltiplas, onde cada fila de execução é associada a um processador, tornando o sistema mais rápido em sistemas SMP (*shared memory multiprocessor machines* - máquina multiprocessadas com memória compartilhada).

Também devem ser realizadas otimizações no código para diminuir o custo do mecanismo.

Pode-se pensar, como trabalho futuro, a implementação de um meta-escalonador utilizado em grades (*grids*[1]) computacionais a partir desta proposta. Os meta-escalonadores centralizam o controle dos processos e dos recursos da grade, permitindo aos usuários submeterem processos para serem executados na grade computacional [Mau05]. O meta-escalonador determina onde o processo irá executar de acordo com as políticas da organização e o nível de serviço requerido. Para garantir a boa utilização dos recursos, o meta-escalonador deve dinamicamente restringir ou privilegiar a utilização dos recursos em cada computador da grade computacional. Essas características são encontradas neste modelo de gestão de recursos.

Este modelo permite que meta-escalonadores sejam mais facilmente implementados, visto que as primitivas necessárias para o desenvolvimento desses *softwares* já estão implementadas.

Além de um desenvolvimento mais aprimorado do modelo, a maior perspectiva é a incorporação do modelo ao núcleo do sistema Linux e de outros sistemas operacionais.

[1] Grades são plataformas para computação geograficamente distribuídas. Elas provêm poder computacional acima de qualquer outro sistema de computação paralela através da união dos recursos físicos de diferentes computadores em um único recurso virtual [Ski02].

Referências Bibliográficas

[Bal02] Leonardo Balliache. Quality of service. Na internet: http://opalsoft.net/qos/QOS.htm, September 2002.

[Bar97] Paul Barham. A fresh approach to file system quality of service. In *7th International Workshop on Network and Operational Systems Support for Digital Audio and Video*. IEEE, 1997.

[BB96] H. Bentaleb and C. Bétourné. QoS generic mechanisms in an operating system. In *WFCS'97 IEEE International Workshop on Factory Communication Systems*, pages 49–58. IEEE, 1996.

[BC00] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, 2000.

[CI01] Stephen Childs and David Ingram. The Linux-SRT integrated multimedia operating system: bringing QoS to the desktop. In *RTAS'01 IEEE Real Time Technology and Applications Symposium*. IEEE, 2001.

[dAMF03] Diego de Assis Monteiro Fernandes. Resposta automática em um sistema de segurança imunológico computacional. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, 2003.

[Dei05] Borislav Deianov. Hierarchical fair scheduler for linux. http://fairsched.sourceforge.net/, Jun 2005.

[DNFW02] Will Dinkel, Douglas Niehaus, Michael Frisbie, and Jacob Woltersdorf. *KURT Linux user manual*, 2002.

[dOCT01] Rômulo de Oliveira, Alexandre Carissimi, and Simão Toscani. *Sistemas Operacionais*. Editora Sagra-Luzzato, 2001.

[DS96] Michael B. Davis and Jaroslaw J. Sydir. Position paper: Resource management for complex distributed system. In *2nd Workshop on Object-Oriented Real-Time Dependable Systems*, pages 113–115, 1996.

[FJ95] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *ACM Transations on Network*, 3(4), August 1995.

[FM02] Ida M. Flynn and Ann McIver McHoes. *Introdução aos Sistemas Operacionais*. Pioneira Thompson Learning, 1 edition, 2002.

[Fou01] Beet Foundation. The free on-line dictionary of computing. Na internet: http://www.beetfoundation.com, 2001.

[Gol05] Karol Golab. Cpu - cap processor usage. Na internet: http://rshk.co.uk/projects/cpucap/, 2005. TLS Technologies.

[GR02] Sourav Glosh and Ragunathan Rajkumar. Resource management of the OS network subsystem. In *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE, 2002.

[GV96] Pawan Goyaland Xingang Guo and Harrick M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *2nd Symp. on Operating Systems Design and Implementation*, Oct 1996.

[Jac02] Jim Jackson. Multiprocessor soft real time cpu resource manager and its validation with hypermedia applications. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2002.

[Kar01] Jan Kara. *Disk Quota System for Linux*, 2001. Na internet: http://doc.alejandro.codina.nom.br/quota-3.08/quotadoc.html.

[Lib01] GNU C Library. Process group functions. Na internet: http://www.gnu.org, May 2001.

[Lov02] Robert Love. Robert love explains variable hz. Na internet: http://kerneltrap.org/node/464, October 2002.

[Lov03] Robert Love. *Linux Kernel Development*. Sams, 1 edition, Sep 2003.

[Mau05] Jeff Mausolf. Grid in action: Managing the resource managers. *Developerworks - IBM*, Jul 2005. Na internet: http://www-128.ibm.com/developerworks/grid/library/gr-metasched/.

[Mol05] Ingo Molnar. Ultra-scalable o(1) smp and up scheduler. http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.0/0810.html, Jun 2005.

[Mon02] Mário A. Monteiro. *Introdução à Organização de Computadores*. LTC, 4 edition, 2002.

[MR95] C. W. Mercer and R. Rajkumar. An interactive interface and RT-Mach support for monitoring and controlling resource management. In *RTAS'01 IEEE Real-Time Technology and Applications Symposium*, pages 134–139. IEEE, 1995.

[MS96] Larry McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Winter USENIX Conference*, 1996.

[NBB97] Hong Quang Nguyen, Christian Bac, and Guy Bernard. Integrating QoS management in a micro-kernel based UNIX operating system. In *Proceedings of the 23rd Euromicro Conference*, pages 371–378. IEEE, 1997.

[NFC+03] Shailabh Nagar, Hubertus Franke, Jonghyuk Choi, Chandra Seetharaman, Scott Kaplan, Nivedita Shinghvi, Vivek Kashyap, and Mike Kravetz. Class-based prioritized resource control in linux. *Ottawa Linux Symposium*, 2003.

[Noe98] GJ Noer. Cygwin: A free win32 porting layer for unix applications. In *2d USENIX NT Symposium*, 1998.

[NvRF+04a] Shailabh Nagar, Rik van Riel, Hubertus Franke, Chandra Seetharaman, and Vivek Kashyap. Advanced workload management support for linux. *LinuxTag*, 2004.

[NvRF+04b] Shailabh Nagar, Rik van Riel, Hubertus Franke, Chandra Seetharaman, Vivek Kashyap, and Haoqiang Zheng. Improving linux resource control using CKRM. In *Proceedings of the Linux Symposium*, volume 2, pages 511–524, 2004.

[OCT02] Rômulo Oliveira, Alexandre Carissimi, and Simão Toscani. Organização de sistemas operacionais convencionais e de tempo real. *XXI JAI, XXII Congresso da SBC*, 2002.

[Pro99a] Linux Documentation Project. *Linux Programmer's Manual*, 1999. setrlimit Manual Page.

[Pro99b] Linux Documentation Project. *Linux Programmer's Manual*, 1999. NanoSleep Manual Page.

[Roe99] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of XIII LISA'99*, 1999.

[SCG+00] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin. Application performance in the qlinux multimedia operating system. In *Proceedings of the Eighth ACM Conference on Multimedia*, Nov 2000.

[SGG02] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Text Books, 2002.

[Ski02] D.B. Skillicorn. Motivating computational grids. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 2002.

[SMJ04] Márcio Starke, Carlo Maziero, and Edgard Jamhour. Controle dinâmico de recursos em sistemas operacionais. In *Sociedade Brasileira de Computação, Anais do I Workshop de Sistemas Operacionais*, volume 1, pages 1–10, Porto Alegre, RS, 2004.

[SNK05] Chandra Seetharaman, Shailabh Nagar, and Vivek Kashyap. Design details of CKRM. Na internet: http://ckrm.sourceforge.net/ckrm_design.htm, feb 2005.

[Tro99] Erik Troan. The linux file access primitives. *Linux Magazine*, 1999.

[TW97] Andrew S. Tannenbaum and Albert S. Woodhull. *Sistemas Operacionais: Projeto e Implementação*. Bookman, 1997.

[Var96] Uresh Varalia. *Unix Internals: The new frontiers*. Prentice Hall, 1996.

[YC01] David K. Y. Yau and Xiangjing Chen. Resource management in software programmable router operating systems. *IEEE Journal on Selected Areas in Communications*, 19(3), March 2001.

[Yod99] Victor Yodaiken. The rtlinux manifesto. In *Proc. of The 5th Linux Expo*, 1999.

[ZDE+93] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new Resource ReSerVation Protocol. IEEE Network, Sep 1993.

Apêndice A

Código fonte do protótipo

```
diff -Naur linux-2.4.21/arch/i386/kernel/entry.S linux-2.4.21-rl/arch/i386/kernel/entry.S
--- linux-2.4.21/arch/i386/kernel/entry.S        2003-06-13 11:51:29.000000000 -0300
+++ linux-2.4.21-rl/arch/i386/kernel/entry.S     2004-07-28 17:48:00.000000000 -0300
@@ -663,6 +663,7 @@
         .long SYMBOL_NAME(sys_ni_syscall)        /* sys_epoll_wait */
         .long SYMBOL_NAME(sys_ni_syscall)        /* sys_remap_file_pages */
         .long SYMBOL_NAME(sys_ni_syscall)        /* sys_set_tid_address */
+        .long SYMBOL_NAME(sys_resourcelimit)     /* 259 sys_resourcelimit marcio@ppgia.pucpr.br */

         .rept NR_syscalls-(.-sys_call_table)/4
                 .long SYMBOL_NAME(sys_ni_syscall)
diff -Naur linux-2.4.21/arch/i386/kernel/init_task.c linux-2.4.21-rl/arch/i386/kernel/init_task.c
--- linux-2.4.21/arch/i386/kernel/init_task.c    2001-09-17 19:29:09.000000000 -0300
+++ linux-2.4.21-rl/arch/i386/kernel/init_task.c         2004-07-29 14:04:58.000000000 -0300
@@ -6,11 +6,15 @@
 #include <asm/pgtable.h>
 #include <asm/desc.h>

+#define __INIT_RLGROUPS
+#include <linux/resourcelimits.h>
+
 static struct fs_struct init_fs = INIT_FS;
 static struct files_struct init_files = INIT_FILES;
 static struct signal_struct init_signals = INIT_SIGNALS;
 struct mm_struct init_mm = INIT_MM(init_mm);

+
 /*
  * Initial task structure.
  *
@@ -31,3 +35,10 @@
  */
 struct tss_struct init_tss[NR_CPUS] __cacheline_aligned = { [0 ... NR_CPUS-1] = INIT_TSS };

+
+/*resourcelimits*/
+unsigned long RLNEXT = 0;
+struct list_head rlgroups;
+int need_recalculate = 0;
+int there_is_lower_rules = 0;
+/*resourcelimits*/
diff -Naur linux-2.4.21/include/asm-i386/unistd.h linux-2.4.21-rl/include/asm-i386/unistd.h
--- linux-2.4.21/include/asm-i386/unistd.h       2002-11-28 21:53:15.000000000 -0200
```

```
+++ linux-2.4.21-rl/include/asm-i386/unistd.h    2004-07-28 17:48:00.000000000 -0300
@@ -257,6 +257,7 @@
 #define __NR_alloc_hugepages    250
 #define __NR_free_hugepages     251
 #define __NR_exit_group                 252
+#define __NR_resourcelimit      259 /* marcio@ppgia.pucpr.br resourcelimits */

 /* user-visible error numbers are in the range -1 - -124: see <asm-i386/errno.h> */

diff -Naur linux-2.4.21/include/linux/resourcelimits.h linux-2.4.21-rl/include/linux/resourcelimits.h
--- linux-2.4.21/include/linux/resourcelimits.h 1969-12-31 21:00:00.000000000 -0300
+++ linux-2.4.21-rl/include/linux/resourcelimits.h        2004-07-30 13:45:43.000000000 -0300
@@ -0,0 +1,40 @@
+#ifndef _LINUX_RESOURCELIMITS_H
+#define _LINUX_RESOURCELIMITS_H
+
+#include <linux/list.h>
+
+#define L_UPPER    1
+#define L_LOWER    2
+
+#define L_CPU      3
+#define L_VM      4
+
+#define RLPROCESS 1
+#define RLUSER 2
+#define RLUSERSGROUP 3
+#define RLPROCESSGROUP 4
+#define RLSESSION 5
+
+#define KEEP_RUNNING 1000
+#define HALF_KEEP_RUNNING 500
+
+struct TRLGroup {
+        int resource;
+        int scope;
+        int target;
+         long value;
+         int type;
+        int RLUSEDTICKS;
+        int active;
+        int number_of_elements;
+        struct list_head list;
+};
+
+#ifndef __INIT_RLGROUPS
+extern unsigned long RLNEXT;
+extern struct list_head rlgroups;
+extern int need_recalculate;
+extern int there_is_lower_rules;
+#endif
+
+#endif
diff -Naur linux-2.4.21/include/linux/sched.h linux-2.4.21-rl/include/linux/sched.h
--- linux-2.4.21/include/linux/sched.h  2003-06-13 11:51:39.000000000 -0300
+++ linux-2.4.21-rl/include/linux/sched.h        2004-07-30 13:45:43.000000000 -0300
@@ -108,6 +108,7 @@
 #define SCHED_OTHER              0
 #define SCHED_FIFO               1
 #define SCHED_RR                 2
+#define SCHED_RL                 3 /* resourcelimits*/

 /*
  * This is an additional bit set when we want to
@@ -415,6 +416,11 @@
```

```
 /* journalling filesystem info */
        void *journal_info;
+
+/* resourcelimits */
+/* marcio@ppgia.pucpr.br 11/11/2003 */
+        struct TRLGroup *rlgroup;
+/* end resourcelimits */
 };

 /*
@@ -435,6 +441,7 @@

 #define PF_USEDFPU      0x00100000      /* task used FPU this quantum (SMP) */

+
 /*
  * Ptrace flags
  */
@@ -509,6 +516,8 @@
     blocked:            {{0}},                                  \
     alloc_lock:         SPIN_LOCK_UNLOCKED,                     \
     journal_info:       NULL,                                   \
+/* resourcelimits */                                            \
+    rlgroup: NULL                                              \
 }


diff -Naur linux-2.4.21/include/linux/unistd.h linux-2.4.21-rl/include/linux/unistd.h
--- linux-2.4.21/include/linux/unistd.h 2001-10-17 15:24:23.000000000 -0200
+++ linux-2.4.21-rl/include/linux/unistd.h      2004-07-28 17:48:00.000000000 -0300
@@ -8,4 +8,5 @@
  */
 #include <asm/unistd.h>

+
 #endif /* _LINUX_UNISTD_H_ */
diff -Naur linux-2.4.21/kernel/exit.c linux-2.4.21-rl/kernel/exit.c
--- linux-2.4.21/kernel/exit.c  2002-11-28 21:53:15.000000000 -0200
+++ linux-2.4.21-rl/kernel/exit.c       2004-07-29 17:01:02.000000000 -0300
@@ -20,6 +20,7 @@
 #include <asm/uaccess.h>
 #include <asm/pgtable.h>
 #include <asm/mmu_context.h>
+#include <linux/resourcelimits.h>

 extern void sem_exit (void);
 extern struct task_struct *child_reaper;
@@ -424,8 +425,11 @@

 NORET_TYPE void do_exit(long code)
 {
-       struct task_struct *tsk = current;
-
+       struct task_struct *tsk = current, *p;
+       int more_tasks = 0;
+       struct list_head *head;
+       struct TRLGroup *node;
+
        if (in_interrupt())
                panic("Aiee, killing interrupt handler!");
        if (!tsk->pid)
@@ -456,6 +460,26 @@
        if (tsk->binfmt && tsk->binfmt->module)
                __MOD_DEC_USE_COUNT(tsk->binfmt->module);
```

```
+       if (tsk->policy == SCHED_RL){
+               for_each_task(p){
+                       if (p->rlgroup == tsk->rlgroup && p != tsk)
+                               more_tasks = 1;
+               }
+               if (!more_tasks){
+                       list_for_each(head, &rlgroups){
+                               node = list_entry(head, struct TRLGroup, list);
+                               if (tsk->rlgroup->scope == node->scope && tsk->rlgroup->target == node
    ->target){
+                                       if (node->type == L_LOWER)
+                                               there_is_lower_rules--;
+//                                     list_del(&(node->list));
+                               }
+                       }
+                       kfree(tsk->rlgroup);
+               }
+               else
+                       tsk->rlgroup->number_of_elements--;
+       }
+
        tsk->exit_code = code;
        exit_notify();
        schedule();
diff -Naur linux-2.4.21/kernel/fork.c linux-2.4.21-rl/kernel/fork.c
--- linux-2.4.21/kernel/fork.c  2003-06-13 11:51:39.000000000 -0300
+++ linux-2.4.21-rl/kernel/fork.c       2004-07-29 17:04:08.000000000 -0300
@@ -28,6 +28,7 @@
 #include <asm/uaccess.h>
 #include <asm/mmu_context.h>
 #include <asm/processor.h>
+#include <linux/resourcelimits.h>

 /* The idle threads do not count.. */
 int nr_threads;
@@ -742,6 +743,18 @@
        if (!current->counter)
                current->need_resched = 1;

+       /* resourcelimit */
+       if (current->policy == SCHED_RL){
+               if (current->rlgroup->scope == RLPROCESS){
+                       current->policy = SCHED_OTHER;
+                       current->rlgroup = NULL;
+               }
+               else
+                       current->rlgroup->number_of_elements++;
+       }
+       /* end resourcelimit */
+
+
        /*
         * Ok, add it to the run-queues and make it
         * visible to the rest of the system.
diff -Naur linux-2.4.21/kernel/sched.c linux-2.4.21-rl/kernel/sched.c
--- linux-2.4.21/kernel/sched.c 2003-06-13 11:51:39.000000000 -0300
+++ linux-2.4.21-rl/kernel/sched.c      2004-07-29 17:03:31.000000000 -0300
@@ -29,6 +29,7 @@
 #include <linux/completion.h>
 #include <linux/prefetch.h>
 #include <linux/compiler.h>
+#include <linux/resourcelimits.h>

 #include <asm/uaccess.h>
```

```
 #include <asm/mmu_context.h>
@@ -149,14 +150,37 @@
          * select the current process after every other
          * runnable process, but before the idle thread.
          * Also, dont trigger a counter recalculation.
-         */
+        */
         weight = -1;
-        if (p->policy & SCHED_YIELD)
-                goto out;

         /*
          * Non-RT process - normal case first.
          */
+
+        if (p->policy == SCHED_RL){
+                /* resourcelimits */
+                weight = p->counter;
+                if (!need_recalculate){
+                        if ( p->rlgroup->type == L_UPPER){
+                                if (! p->rlgroup->RLUSEDTICKS)
+                                        weight=-1001;
+                         }
+                         else
+                                if (p->rlgroup->RLUSEDTICKS){
+                                        weight += (KEEP_RUNNING + p->counter);
+                                }
+                                else{
+                                        p->counter = NICE_TO_TICKS(20); //menor prioridade
+                                        weight = p->counter;
+                                }
+                 }
+                 goto out;
+        }
+
+        if (p->policy & SCHED_YIELD)
+                goto out;
+
+
         if (p->policy == SCHED_OTHER) {
                 /*
                  * Give the process a first-approximation goodness value
@@ -165,9 +189,15 @@
                  * Don't do any other calculations if the time slice is
                  * over..
                  */
+
                 weight = p->counter;
+
                 if (!weight)
                         goto out;
+
+
+                /*end resourcelimit*/
+

 #ifdef CONFIG_SMP
                 /* Give a largish advantage to the same processor...    */
@@ -571,9 +601,9 @@
          * only one process per CPU.
          */
         sched_data = & aligned_data[this_cpu].schedule_data;
-
+
         spin_lock_irq(&runqueue_lock);
```

```
-
+
        /* move an exhausted RR process to be last.. */
        if (unlikely(prev->policy == SCHED_RR))
                if (!prev->counter) {
@@ -613,13 +643,23 @@
        }

        /* Do we need to re-calculate counters? */
-       if (unlikely(!c)) {
+       if (unlikely(!c) || need_recalculate) {
                struct task_struct *p;
-
                spin_unlock_irq(&runqueue_lock);
                read_lock(&tasklist_lock);
-               for_each_task(p)
-                       p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
+               for_each_task(p){
+                       if (p->policy != SCHED_RL){
+                               p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
+                       }
+                       else {
+                               if (need_recalculate) {
+                                       p->rlgroup->RLUSEDTICKS = p->rlgroup->value;
+                                       p->counter = (p->rlgroup->RLUSEDTICKS / p->rlgroup->
    number_of_elements);
+                               }
+                       }
+
+               }
+               need_recalculate = 0;
                read_unlock(&tasklist_lock);
                spin_lock_irq(&runqueue_lock);
                goto repeat_schedule;
@@ -636,7 +676,8 @@

        if (unlikely(prev == next)) {
                /* We won't go through the normal tail, so do this by hand */
-               prev->policy &= ~SCHED_YIELD;
+               if (prev->policy != SCHED_RL)
+                       prev->policy &= ~SCHED_YIELD;
                goto same_process;
        }

@@ -1133,6 +1174,7 @@
        case SCHED_RR:
                ret = 99;
                break;
+       case SCHED_RL:
        case SCHED_OTHER:
                ret = 0;
                break;
@@ -1149,6 +1191,7 @@
        case SCHED_RR:
                ret = 1;
                break;
+       case SCHED_RL:
        case SCHED_OTHER:
                ret = 0;
        }
diff -Naur linux-2.4.21/kernel/sys.c linux-2.4.21-rl/kernel/sys.c
--- linux-2.4.21/kernel/sys.c   2003-06-13 11:51:39.000000000 -0300
+++ linux-2.4.21-rl/kernel/sys.c        2004-07-30 13:46:58.000000000 -0300
@@ -14,6 +14,7 @@
 #include <linux/prctl.h>
```

```
 #include <linux/init.h>
 #include <linux/highuid.h>
+#include <linux/resourcelimits.h>

 #include <asm/uaccess.h>
 #include <asm/io.h>
@@ -1286,6 +1287,116 @@
        return error;
 }

+
+asmlinkage long sys_resourcelimit(int resource, int scope, int target, long value, int type)
+{
+       int cpu_used = 0;
+        struct task_struct *p;
+       struct TRLGroup *node;
+       static int FIRST_TIME = 1;
+
+       if (FIRST_TIME){
+               FIRST_TIME = 0;
+               INIT_LIST_HEAD(&rlgroups);
+       }
+
+
+        if( value < 0 || value > 100)
+                return -EINVAL;
+
+        /* normalize value */
+        if (value < 1)
+                       value = 1;
+        if (value > 100)
+                value = 100;
+
+       /* verifica limites de utilizacao Maxima 100%*/
+       for_each_task(p){
+               if(p->policy == SCHED_RL && p->rlgroup->type == L_LOWER)
+                       cpu_used += p->rlgroup->value;
+       }
+
+        if (cpu_used + value >= 100)
+               return -EINVAL;
+
+        /* convert to jiffies, check limits as a precaution */
+        value = (value * HZ) / 100;
+        if (value < 1)
+               value = 1;
+        if (value > HZ)
+               return -EINVAL;
+
+
+       node = (struct TRLGroup *) kmalloc(sizeof(struct TRLGroup),GFP_KERNEL);
+
+       node->value = value;
+       node->type = type;
+       node->scope =  scope;
+       node->RLUSEDTICKS = -1;
+       node->active = 1;
+       node->target = target;
+       node->resource = L_CPU;
+       node->number_of_elements = 0;
+
+       list_add(&node->list, &rlgroups);
+/* POR PROCESSO */
+       if (scope == RLPROCESS){
+               p = find_task_by_pid(target);
```

```
+               if (!p)
+                       return -ESRCH;
+               read_lock(&tasklist_lock);
+               p->policy = SCHED_RL;
+               p->rlgroup = node;
+               p->rlgroup->number_of_elements++;
+               read_unlock(&tasklist_lock);
+
+       }
+
+/* USUARIO */
+       if (scope == RLUSER){
+               for_each_task(p){
+                       if (p->uid == target){
+                               read_lock(&tasklist_lock);
+                               p->policy = SCHED_RL;
+                               p->rlgroup = node;
+                               p->rlgroup->number_of_elements++;
+                               read_unlock(&tasklist_lock);
+                        }
+               }
+       }
+
+/* GRUPO */
+
+        if (scope == RLPROCESSGROUP){
+                for_each_task(p){
+                               if (target == p->gid){
+                                       read_lock(&tasklist_lock);
+                                       p->policy = SCHED_RL;
+                                       p->rlgroup = node;
+                                       p->rlgroup->number_of_elements++;
+                                       read_unlock(&tasklist_lock);
+                               }
+                }
+        }
+
+/* GRUPO DE GRUPO */
+        if (scope == RLSESSION){
+                for_each_task(p){
+                               if (target == p->session){
+                                       read_lock(&tasklist_lock);
+                                       p->policy = SCHED_RL;
+                                       p->rlgroup = node;
+                                       p->rlgroup->number_of_elements++;
+                                       read_unlock(&tasklist_lock);
+                               }
+                }
+        }
+
+       if (type == L_LOWER)
+               there_is_lower_rules++;
+        return 0;
+}
+
 EXPORT_SYMBOL(notifier_chain_register);
 EXPORT_SYMBOL(notifier_chain_unregister);
 EXPORT_SYMBOL(notifier_call_chain);
diff -Naur linux-2.4.21/kernel/timer.c linux-2.4.21-rl/kernel/timer.c
--- linux-2.4.21/kernel/timer.c 2002-11-28 21:53:15.000000000 -0200
+++ linux-2.4.21-rl/kernel/timer.c       2004-07-29 16:52:57.000000000 -0300
@@ -22,6 +22,7 @@
 #include <linux/smp_lock.h>
 #include <linux/interrupt.h>
 #include <linux/kernel_stat.h>
```

```
+#include <linux/resourcelimits.h>

 #include <asm/uaccess.h>

@@ -581,6 +582,23 @@
 void update_one_process(struct task_struct *p, unsigned long user,
                        unsigned long system, int cpu)
 {
+        /* resourcelimit */
+         if (RLNEXT < jiffies){
+                RLNEXT = jiffies + HZ;
+                need_recalculate = 1;
+                if (there_is_lower_rules)
+                        p->counter = 0;
+        }
+        if (p->policy == SCHED_RL){
+                if (--p->rlgroup->RLUSEDTICKS == -1) {
+                        struct task_struct *pp;
+                        p->rlgroup->RLUSEDTICKS=0;
+                        for_each_task(pp){
+                                if (pp->rlgroup == p->rlgroup)
+                                        pp->counter = 0;
+                        }
+                }
+        }
        p->per_cpu_utime[cpu] += user;
        p->per_cpu_stime[cpu] += system;
        do_process_times(p, user, system);
@@ -873,4 +891,3 @@
        }
        return 0;
 }
-
```

Apêndice B

Código fonte do monitor de uso da CPU

```
#include<stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/sysinfo.h>
#include <string.h>
#include <locale.h>
#include <asm/param.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>

#define STAT "/proc/stat"
#define PROCESS_STAT "stat"
#define TRUE 1
#define BUFFER 256

unsigned long long Hertz=0;
int fd_stat, fd_process_stat;
struct sigaction sact;

/* executar quando receber sinal para sair */
void sair(int signo){
        close(fd_stat);
        close(fd_process_stat);
                printf("Saindo...\n");
                exit(0);
}

int main(int argn, char *argv[]){
        int  pid;
        char cpid[80], filename[80];
        struct timespec Sleeptime = {tv_sec: 1, tv_nsec: 0};
        unsigned long T = 0;
        char buf[BUFFER];

        unsigned long oldjiffies = 0, jiffies = 0, user_j, nice_j, sys_j, other_j; /* jiffies (clock
             ticks) */
        unsigned long utime, stime, total_time = 0, old_total_time = 0;
        float pcpu;
```

```
	if( !(argn >=  2 && argn <= 4) ){
		printf("Uso: %s PID [-t delay]\n", argv[0]);
		return 1;
	}

	if( (argn == 4) && (argv[2][1] == 't')){
		Sleeptime.tv_sec = atoi(argv[3]);
	}

	/* manipulador de sinais */
	sact.sa_handler = sair;
	sact.sa_flags = 0;
	sigemptyset(&sact.sa_mask);
	sigaction(SIGHUP, &sact, NULL);
	sigaction(SIGINT, &sact, NULL);
	sigaction(SIGQUIT, &sact, NULL);


	Hertz = (unsigned long long)HZ;
	strcpy(cpid, argv[1]);
	pid = atoi(argv[1]);
	sprintf(filename, "/proc/%s/%s", cpid, PROCESS_STAT);

	fd_process_stat = open(filename, O_RDONLY | O_SYNC);
	fd_stat = open(STAT, O_RDONLY | O_SYNC);

	if ( (fd_process_stat == -1) || (fd_stat == -1)){
		perror("Erro");
		return 1;
	}



	while ( TRUE ){
		read(fd_process_stat, (void *) buf, sizeof(buf));
		sscanf(buf ,"%*lu %*s %*c %*lu %*lu %*lu %*lu %*lu %*lu %*lu %*lu %*lu %*lu %lu %lu",&
		    utime,&stime);
		read(fd_stat, (void *) buf, sizeof(buf));
		sscanf(buf,"cpu %lu %lu %lu %lu\n",&user_j,&nice_j,&sys_j,&other_j);

		total_time = utime + stime;

		jiffies = user_j + nice_j + sys_j + other_j;

		if (T){
			pcpu = (float) ((total_time - old_total_time)*100.  / (float)(jiffies -
			    oldjiffies));
			printf("Usando em T%d: %.2f %%\n", T, pcpu);
		}
		oldjiffies = jiffies;
		old_total_time = total_time;
		T++;
		lseek(fd_process_stat,0,SEEK_SET);
		lseek(fd_stat,0,SEEK_SET);
		fflush(stdout);
		nanosleep(&Sleeptime, NULL);

	}
}
```

Apêndice C

Código fonte do manipulador do protótipo

C.1 resourcelimits.h

```
#ifndef _LINUX_RESOURCELIMITS_H
#define _LINUX_RESOURCELIMITS_H
#define L_UPPER    1
#define L_LOWER    2
#define L_CPU      3
#define L_VM       4
#define RLPROCESS 1
#define RLUSER 2
#define RLUSERSGROUP 3
#define RLPROCESSGROUP 4
#define RLSESSION 5

#define resourcelimit(__resource, __scope, __target, __value, __type) \
        syscall(259, __resource, __scope, __target, __value, __type)

#define __NR_resourcelimit         259
#endif
```

C.2 chlimits.c

```
#include<stdio.h>
#include<sys/syscall.h>
#include<string.h>
#include<stdlib.h>
#include<linux/unistd.h>

#include "resourcelimits.h"

int main(int argn, char *argv[]){
        unsigned long value, target, scope, type;

        if (argn < 4){
          printf("Usage: chlimits scope target %% UPPER|LOWER\n");
          return 0;
```

```
        }
        value = atoi(argv[3]);
        target  = atoi(argv[2]);
        if (strcasecmp("upper",argv[4]) == 0)
                type = L_UPPER;
        else
                type = L_LOWER;
        if (strcasecmp("usuario",argv[1]) == 0)
                scope = RLUSER;
        else if (strcasecmp("grupo",argv[1]) == 0)
                scope = RLPROCESSGROUP;
             else if (strcasecmp("processo",argv[1]) == 0)
                        scope = RLPROCESS;
                  else
                        scope = RLSESSION;

        if (value < 0 || value > 100){
                printf("Erro...\n");
                return 0;
        }
        printf("Pid: %d %% %d \n", target, value);
        resourcelimit(L_CPU, scope, target, value, type);
        return 0;
}
```