ANTONIO ADÍLSON FERREIRA

Pela lente do CFS: Um olhar sobre o escalonador do Linux

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Redes e Sistemas Distribuídos e Paralelos.

Orientador: Prof. Dr. Vinod Rebello
Coorientadora: Profª Drª. Aline de Paula Nascimento

Niterói
2012

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

F383 Ferreira, Antonio Adílson

Pela lente do CFS : um olhar sobre o escalonador Linux / Antonio Adílson Ferreira. – Niterói, RJ : [s.n.], 2012.
133 f.

Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2012.
Orientadores: Vinod Rebello, Aline de Paula Nascimento .

2 Escalonamento de tarefas. 2. Algoritmo de escalonamento. 3. Sistema Linux. 4. Sistemas paralelos e distribuídos. I. Título.

CDD 005.136

ANTONIO ADÍLSON FERREIRA

Pela lente do CFS: Um olhar sobre o escalonador do Linux

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Redes e Sistemas Distribuídos e Paralelos.

Aprovada em julho de 2012

BANCA EXAMINADORA

Prof. Dr. Eugene Francis Vinod Rebello – Orientador

UFF

Profa. Dra. Aline de Paula Nascimento – Coorientadora

UFF

Prof. Dr. Gabriel Pereira da Silva

UFRJ

Profa. Dra. Maria Cristina da Silva Boeres

UFF

Niterói

2012

Ao meu pai,

in memoriam

AGRADECIMENTOS

Aos meus pais.

À minha mulher e meus filhos.

A PETROBRAS S/A pelo estímulo e concessão para a conclusão deste Mestrado com disponibilização de recursos técnicos e tempo para o mesmo.

Aos colegas da PETROBRAS que me ajudaram em muitas instalações do *kernel* do Linux.

A todos os meus professores do CEDERJ em reconhecimento aos seus valores.

Ao meu orientador D.Sc. Vinod por sua tolerância e boa vontade para comigo, bem como sua significativa contribuição no campo das ideias e compreensão do objetivo do trabalho.

À minha coorientadora D.Sc. Aline, pela enorme ajuda nos momentos difíceis e por sempre manter-me motivado para a conclusão do trabalho.

RESUMO

Em arquiteturas orientadas a serviços, tais como computação em nuvem, Acordos de Nível de Serviço (SLAs) são frequentemente solicitados como garantia de que o pedido será executado conforme programado. Para evitar qualquer violação do SLA, os sistemas tendem a ser pouco utilizado, tornando-os ineficazes. Com o advento de servidores cada vez mais poderosos, com múltiplos processadores *multicore*, por exemplo, a tendência para consolidar servidores e compartilhar recursos entre vários aplicativos simultaneamente só vai continuar a crescer e, por sua vez, tornar a gestão de SLAs ainda mais difícil. Os escalonadores de processos são os principais componentes dos sistemas operacionais modernos. Esta dissertação apresenta os principais escalonadores do Linux e, em detalhes, a sua mais recente encarnação, o *Completely Fair Scheduler* (***CFS***). O trabalho identifica as inovações importantes, como a utilização de uma estrutura de árvore binária, o conceito de *vruntime*, e o uso de temporizadores de granularidade fina com uma precisão de nanossegundos. Através de avaliações experimentais, este trabalho conclui que o ***CFS*** é menos injusto do que o seu antecessor, ***O(1)***. Usando os mecanismos de controle estatístico e de depuração, também é investigada a relação dinâmica dos parâmetros do ***CFS*** para melhorar o desempenho das aplicações. Também é examinada a habilidade do kernel para dividir e compartilhar recursos específicos, como CPU, memória, largura de banda, através da reutilização da ferramenta dos grupos de controle. Finalmente, mostramos a conseqüência direta da tendência do novo kernel *dyntick* - que obtém uma maior economia de energia quando em um estado inativo, e pode ser essencial para otimizar o consumo de dispositivos de energia como celulares, *notebooks* e servidores. Este trabalho conclui que é necessário adotar mecanismos para ajustar os parâmetros do kernel dinamicamente para tornar o escalonador do kernel mais eficiente para aplicações do tipo HPC ou Computação em Nuvem. Além disso, através de grupos de controle, o controle fino da distribuição e contabilização dos recursos, por exemplo, será fundamental para modelos mais justos de custos na computação em nuvem e na implementação e cumprimento de SLAs.

Palavras-chave: escalonador, ***CFS***, *kernel* do Linux, *multicore*, sistema operacional.

ABSTRACT

In service-oriented architectures, such as cloud computing, Service Level Agreements (SLAs) are often requested as a guarantee that an application will be run on schedule. To avoid any breach of the SLA, systems tend to be underutilized, making them inefficient. With the advent of increasingly powerful servers with multiple multicore processors, for example, the tendency to consolidate servers and share resources among multiple applications simultaneously will only continue to grow and, in turn, make the management of SLAs even more difficult. The process scheduler is a key component of modern operating systems. This dissertation presents the main Linux schedulers and, in detail, its latest incarnation, the Completely Fair Scheduler (***CFS***). The work identifies their major innovations, e.g. the use of a binary tree structure, the concept of *vruntime*, and the use of finegrain timers with a nanosecond precision. Through experimental evaluations, this work concludes that the ***CFS*** is less unfair than it's predecessor, ***O(1)***. Using the mechanisms of statistical control and debugging, the relationship between ***CFS*** parameters to improve the performance of applications is also investigated. Also examined is the ability of the kernel to divide and share specific resources such as CPU, memory, bandwidth, through the reuse of the tool *Control Groups*. Finally, we show the direct consequence of the new trend *dyntick kernel* – which achieves higher power savings when in an *idle* state, and might be essential to optimize energy consumption in mobile devices, notebooks and servers. This work concludes that it is necessary to adopt mechanisms to adjust the kernel parameters dynamically to make the kernel scheduler more efficient for HPC type applications or Cloud Computing. Moreover, through *Control Groups*, the fine control of the distribution of resources and accounting, for example, will be fundamental for fairer cost models in Cloud Computing and implementing and meeting SLAs.

Keywords: scheduler, *CFS*, Linux kernel, multicore, operating system.

LISTA DE ILUSTRAÇÕES

Figura 2-1 Diagrama da fila de execução global × fila de execução única por processador 13
Figura 2-2 Estrutura da fila de processos como lista circular duplamente encadeada 13
Figura 2-3 Estrutura básica da fila de execução de processos como duas listas ativa e expirada indexadas por prioridades [26]. 15
Figura 2-4 Diagrama simplificado do escalonador SD [23] 17
Figura 2-5 Estrutura da fila de execução em árvore rubro-negra. 18
Figura 2-6 Diagrama geral do escalonador *CFS* [26] 20
Figura 3-1 Gráfico do consumo de CPU por processo concorrente. Sem *CFS.* 44
Figura 3-2 Gráfico do consumo de CPU por processo concorrente. Com *CFS.* 44
Figura 3-3 Gráfico do consumo de CPU por processo concorrente. Sem *CFS.* 47
Figura 3-4 Gráfico do consumo de CPU por processo concorrente. Com *CFS.* 47
Figura 3-5 Gráfico do consumo de CPU por processo concorrente. Sem *CFS.* 49
Figura 3-6 Gráfico do consumo de CPU por processo concorrente. Com *CFS.* 49
Figura 3-7 Gráfico do consumo de CPU por processo concorrente. Sem *CFS.* 51
Figura 3-8 Gráfico do consumo de CPU por processo concorrente. Com *CFS.* 51
Figura 3-9 Gráfico do tempo de CPU para 4 processos de 1 segundo. 56
Figura 3-10 Gráfico de tempo de CPU para 4 processos de 1 segundo. 56
Figura 4-1 Compartilhamento de CPU por até 48 processos concorrentes - 1 novo processo é disparado a cada 60 segundos sem *CFS* 66
Figura 4-2 Compartilhamento de CPU por até 48 processos concorrentes - 1 novo processo é disparado a cada 60 segundos com *CFS* 66
Figura 4-3 Gráfico do compartilhamento de CPU por até 100 processos concorrentes 69
Figura 4-4 Gráfico do compartilhamento de CPU por até 100 processos concorrentes 69
Figura 4-5 Gráfico do comportamento médio de consumo de CPU por processo em cada instante de tempo. 73
Figura 4-6 Gráfico do comportamento médio de consumo de CPU por processo em cada instante de tempo 73
Figura 4-7 Compartilhamento de CPU por processo com barra de erro 74

Figura 4-8 Compartilhamento de CPU por processo com barra de erro 74
Figura 4-9 Distribuição de % CPU entre 46 processos sem CFS para 1 CPU. 77
Figura 4-10 Distribuição de % CPU entre 46 processos com CFS para 1 CPU. 77
Figura 4-11 Distribuição de % CPU entre 23 processos sem CFS para 2 CPUs. 78
Figura 4-12 Distribuição de % CPU entre 23 processos com CFS para 2 CPUs. 78
Figura 4-13 Distribuição de % CPU entre 23 processos sem CFS para 4 CPUs. 79
Figura 4-14 Distribuição de % CPU entre 23 processos com CFS para 4 CPUs. 79
Figura 4-15 Distribuição de % CPU entre 23 processos com CFS para 6 CPUs. 80
Figura 4-16 Distribuição de % CPU entre 23 processos com CFS para 6 CPUs. 80
Figura 4-17 Distribuição de % CPU entre 46 processos sem CFS para 12 CPUs. 81
Figura 4-18 Distribuição de % CPU entre 46 processos com CFS para 12 CPUs. 81
Figura 4-19 Gráfico que representa o consumo total de CPU para atingir uma mesma quantidade de tarefas em um intervalo de tempo de relógio pré-definido 83
Figura 4-20 Gráfico que representa o tempo de relógio para conclusão de uma mesma quantidade de tarefas em um intervalo de tempo pré-definido (aplicação CF) 85
Figura 4-21 Tempo total de processos paralelos (CF) por número de processos para um latência mínima com seu máximo valor, 1 segundo 86
Figura 4-22 Gráfico da variação do número de trocas de contexto em função da latência mínima definida com o seu máximo valor (1 segundo). 88
Figura 4-23 Gráfico de variação do número de trocas de contexto de processos paralelos em função do escalonador com opção default e latência mínima máxima do *CFS* 89
Figura 5-1 Comportamento praticamente equânime para cargas distintas de jobs com diferentes usuários 100
Figura 5-2 Gráfico da distribuição de CPU entre Pesquisadores e Estudantes 103
Figura 5-3 Exemplo de partilha de CPU para subgrupos de *CGroups* 105
Figura 5-4 Gráfico da segunda distribuição de CPU por subgrupo 106
Figura 5-5 Comportamento praticamente linear de Jobs com múltiplos processos concorrentes de mesma carga. 109

LISTA DE TABELAS

Tabela 3-1 Variâncias em pequena escala com CF = 1 segundo 45
Tabela 3-2 Variâncias em pequena escala com CF = 1 segundo 48
Tabela 3-3 Variâncias em pequena escala com CF = 10 segundos 50
Tabela 3-4 Variâncias em pequena escala com CF = 10 segundos 52
Tabela 4-1 Distribuição da carga de CPU por *core* no tempo com até 12 processos concorrentes executando em 4 *cores* 61
Tabela 4-2 Preservação e manutenção do processo no próprio *core* p/ 4 *cores* 63
Tabela 4-3 Variação dinâmica de parâmetros do *CFS* 64
Tabela 4-4 Análise da variância em função do número de CPUs com e sem CFS 82
Tabela 5-1 Tempos Médios de Execução de processos concorrentes de carga fixada em 10 x 500.000.000 adições unitárias (~ 10 segundos) por 2 ou 3 Jobs de distintos usuários 99
Tabela 5-2 Execução de processos concorrentes de 60 segundos cada 101
Tabela 5-3 Tempos relativos extraídos da contabilização dos *cgroups* 102
Tabela 5-4 Dados para teste de subgrupos em *CGroups* 105
Tabela 5-5 Um segundo exemplo de distribuição temporal de CPU 106
Tabela 5-6 Tempos Médios de Execução de processos concorrentes de carga fixada em 4 x 500.000.000 adições unitárias por 1 único job de um mesmo usuário 107

LISTA DE ABREVIATURAS E SIGLAS

AT&T	American Telephone and Telegraph
CFQ	Completely Fair Queuing
CFS	Completely Fair Scheduler
CPU	Central Processor Unit
FCFS	First-Come, First-Server
GE	General Eletric
HPC	High Performance Computing
HT	Hyper-Threading
IEEE	Institute of Electrical and Electronics Engineers
ILP	Instruction-Level Parallelism
MIT	Massachussetts Institute of Technology
RDS	Rotating Staircase
RR	Round-Robin
RSDL	Rotating Staircase Deadline
SJF	Shortest Job First
SLA	Service Level Agreement
SMP	Symmetric Multiprocessing
SMT	Simultaneous MultiThreading
TLP	Thread-Level Parallelism

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO 1

1.1 MOTIVAÇÃO 1
1.2 HISTÓRICO 3
1.3 OBJETIVO 7
1.4 ORGANIZAÇÃO DESTA DISSERTAÇÃO 7

CAPÍTULO 2 – ESCALONADORES DE PROCESSOS OU TAREFAS 9

2.1 ESCALONADORES EM GERAL 9
2.2 O ESCALONADOR PRIMORDIAL 11
2.3 O ESCALONADOR *O*(*N*) 12
2.4 O ESCALONADOR *O*(*1*) 13
2.5 O ESCALONADOR RSDL → SD 15
2.6 ESCALONADOR CFS 17
2.6.1 PRINCIPAIS COMPONENTES DO ESCALONADOR *CFS*: 19
2.6.2 AJUSTE FINO DOS PRINCIPAIS PARÂMETROS DO ESCALONADOR *CFS* 22
2.6.3 ALTERAÇÃO EM TEMPO DE EXECUÇÃO DOS PARÂMETROS DINAMICAMENTE RECONFIGURÁVEIS DO ESCALONADOR CFS 23
2.6.4 DETERMINAÇÃO DO *TIME SLICE* PARA CADA PROCESSO NO *CFS* 26
2.7 TRABALHOS CORRELATOS 27
2.8 RESUMO 33

CAPÍTULO 3 – DETALHAMENTO DO CSF 34

3.1 DETALHAMENTO E SÍNTESE DO *CSF* 34
3.1.1 DETALHAMENTO DO FUNCIONAMENTO DO ALGORITMO DO CFS 36
3.2 METODOLOGIA DOS EXPERIMENTOS COM OU SEM O CFS 39
3.2.1 COM *LOOP INFINITO* (QUEIMA INDEFINIDA DE SEGUNDOS DE TEMPO DE CPU) 40
3.2.2 COM CARGA DEFINIDA (QUEIMA DE 500.000.000 ADIÇÕES DE UMA CONSTANTE EM UMA VARIÁVEL INTEIRA) 41
3.3 RESULTADOS PRELIMINARES 42
3.3.1 AVALIAÇÃO DO ESCALONADOR EM UMA PEQUENA ESCALA 43

3.3.2 BALANCEAMENTO DE CARGA (*UNDERPROVISIONING*) 54
3.3.3 IMPACTO DA RECONFIGURAÇÃO DINÂMICA DO *CFS* 55
3.4 RESUMO 57

CAPÍTULO 4 – ANÁLISE EXPERIMENTAL 58

4.1 BALANCEAMENTO DE CARGA (*UNDERPROVISIONING*) 58
4.2 BALANCEAMENTO DE CARGA – EFEITO *PING-PONG* 62
4.3 AUTO-CONFIGURAÇÃO DE PARÂMETROS 63
4.4 AVALIANDO A EQUIDADE DO *CFS* EM LARGA ESCALA – VARIANDO O NÚMERO DE PROCESSOS 65
4.5 AVALIANDO A EQUIDADE DO *CFS* EM LARGA ESCALA – VARIANDO O NÚMERO DE *CORES* 76
4.6 OVERHEAD DO SISTEMA 83
4.6.1 CASO 1 – PROCESSOS LI 83
4.6.2 CASO 2 – PROCESSOS CF 84
4.7 MUDANÇA DA LATÊNCIA MÍNIMA DO ESCALONADOR *CFS* 86
4.7.1 AVALIAÇÃO DO TEMPO DE EXECUÇÃO 86
4.7.2 AVALIAÇÃO DO NÚMERO DE TROCAS DE CONTEXTO EM FUNÇÃO DO TAMANHO DO EXECUTÁVEL 87
4.7.3 AVALIAÇÃO DA TROCA DE CONTEXTO COM E SEM *CFS* 89
4.8 RESUMO 90

CAPÍTULO 5 – GRUPOS DE CONTROLE DO CFS 92

5.1 CGROUPS - CONTROL GROUPS 92
5.1.1 BREVE HISTÓRICO DOS GRUPOS DE CONTROLE 93
5.1.2 *CGROUPS VERSUS* VIRTUALIZAÇÃO 94
5.1.3 FORMALIZANDO CONCEITOS DE *CGROUPS* 95
5.2 ARQUIVOS DE CONFIGURAÇÃO E DAEMONS DO CGROUP 96
5.3 ADMINISTRANDO E MANTENDO CGROUPS 97
5.4 EXPERIMENTOS COM *CGROUPS* 98
5.4.1 DIVISÃO DO TEMPO DE CPU SEM *CGROUPS* 98
5.4.2 DIVISÃO DE TEMPO DE CPU COM *CGROUPS* 101
5.4.3 PARTILHA PONDERADA DE RECURSOS COMPARTILHADOS PARA *CGROUPS* ATIVO 102

5.4.4 PARTILHA PONDERADA DE RECURSOS COMPARTILHADOS PARA *SUBGRUPOS* DE *CGROUPS* 104
5.4.5 LINEARIDADE DO TEMPO DE EXECUÇÃO MÉDIO PARA CARGA FIXA (CF) SEM *CGROUPS* 107
5.5 RESUMO 109

CAPÍTULO 6 – CONCLUSÕES E TRABALHOS FUTUROS 110

APÊNDICE A – 123

ANEXO A – 127

Estrutura de Dados 127
Escalonamento de Tarefas 128
Balanceador de Carga 131
CLASSE DE ESCALONAMENTO *CFS* 132

CAPÍTULO 1 – INTRODUÇÃO

Os sistemas de HPC ou computação distribuída são um excelente nicho de mercado e de significativa contribuição para a humanidade. As aplicações em HPC visam soluções de problemas de física, biologia, química, matemática, geologia, geofísica, previsão de tempo, etc. Estas aplicações, em geral, prescindem de programas que são CPU intensivos e, algumas vezes, com prazos críticos de solução (previsão do tempo, por exemplo). O sistema operacional [1] Linux tem participação em 92,4% do mercado de HPC [2] e, com os sistemas UNIX, detêm mais de 60% do mercado de servidores [3]. Estudos de melhoria de desempenho do Linux para estes ambientes e outros, como o promissor mercado de Computação em Nuvem, por exemplo, são de vital importância econômica e científica. Neste trabalho, estudaremos um dos principais componentes do *kernel* do Linux, o seu novo escalonador ***CFS***, em particular, para que possamos contribuir com sugestões e recomendações para a melhoria do desempenho de aplicações que possam tirar proveito da nova tecnologia de processadores *multicore* [4] [5] [6] [7] [8] [9] [10].

1.1 MOTIVAÇÃO

Nos anos 90, por ser considerado de complexa instalação pelos usuários comuns em suas máquinas, havia uma enorme dificuldade de inserção do Linux no então pouco concorrido território dos sistemas operacionais dos PCs, *desktops* e *workstations*. Apesar das dificuldades de uso para os usuários comuns, o Linux era um software livre e de código aberto, o que despertava o interesse dos aficionados em desenvolver seu próprio código e de parte da comunidade científica e do mundo acadêmico. Entretanto, o mercado de computação para *desktops* era, e continua sendo, por enquanto, dominado de fato pelos sistemas operacionais Microsoft Windows e Apple MacOS (Europa e Estados Unidos, principalmente), cerca de 99% [11]. Inicialmente, o motivo principal era a ausência de uma interface gráfica, além do sentimento de baixa interatividade do Linux associado, também, a um baixo tempo de resposta. A questão da interface gráfica foi resolvida com o KDE 1.0 para GNU/Linux ou BSD, lançado em 12 de julho de 1998 [12]. Outro problema era o desempenho do *kernel* do Linux, segundo entrevista de um dos seus principais desenvolvedores, Con Kolivas [13], em julho de 2007. O problema precisava urgentemente ser resolvido com substanciais alterações fundamentais do escalonador do *kernel* do Linux para permitir um melhor desempenho

e com maior interatividade experimentada pelos usuários de *desktops* e *workstations*, bem como um menor tempo de resposta do sistema como um todo [13] [14].

Na década passada, surgiram as máquinas *multicores*, que passaram a dominar o mercado até os dias atuais. A tecnologia *multicore* foi uma necessidade [6] para se umentar o desempenho dos processadores sem recorrer a aumentos na frequência de processamento o que implica em um maior consumo de energia e maior necessidade de refrigeração da energia dissipada pelo processador. A dissipação de calor, a necessidade de refrigeração e o acesso a memória eram consideradas barreiras para se obter maior desempenho com CPUs monoprocessadas, ou seja, um processador por *chip* físico. As CPUs monoprocessadas só podiam aproveitar o aumento de desempenho com um paralelismo a nível de instrução (ILP) enquanto os processadores *multicores* podem aproveitar o paralelismo também a nível de *thread* ou processo (TLP).

Portanto, estas novas arquiteturas *multicore* passaram a exigir sistemas operacionais mais eficientes e escaláveis, ou seja, capazes de usufruir melhor e obter maior desempenho com a execução de processos simultaneamente. Então, fazia parte das diretrizes dos desenvolvedores do Linux torná-lo mais competitivo. Uma melhoria obrigatória do *kernel* do Linux era fundamental para a sua disseminação e sobrevivência no mercado de *desktops* e *workstations* [13] [14] além dos inúmeros dispositivos embarcados e móveis que se apresentariam no início do século XXI, com o enorme avanço das telecomunicações sem fio, dos tablets, das consoles de jogos, e dos sistemas operacionais embarcados nestes dispositivos (iOS, Android, etc). É importante destacar que o Linux dominava e ainda domina em 60% o mercado de servidores [3], porque era e continua sendo gratuito. Economicamente não havia interesse, mas tecnicamente era um excelente desafio com implicações diretas em nichos de mercado como os de *Clusters*, HPC e Computação em Nuvem (*Cloud Computing*) que dependem do Linux, pois são formados por unidades de composição como *blades*, *workstations*, *servers*, nós de *cluster*, etc, com o Linux instalado.

Neste trabalho é estudado o comportamento do escalonamento de processos no nível do *kernel* do Linux através de seu novo escalonador ***CFS.*** A finalidade principal é estender a compreensão do escalonador para a determinação de um ***QoS*** para um nó, a unidade de processamento, usada em *Clusters*, HPC ou Computação em Nuvem,

permitindo assim uma garantia na qualidade do serviço e, por exemplo, uma tarifação[1] justa do cômputo do consumo de CPU ou outros recursos computacionais pelas aplicações.

1.2 HISTÓRICO

Nos anos 60 foi desenvolvido no MIT um projeto de sistema operacional Multics em conjunto com a GE, Laboratórios Bell e AT&T. O objetivo era que o Multics tivesse características de um sistema operacional de tempo compartilhado, isto é, vários usuários poderiam compartilhar os recursos de um único computador. Ele foi totalmente desenvolvido em linguagem de máquina (*assembly*). Após a retirada do Bell Labs do projeto, um dos seus pesquisadores, Ken Thompson, deu continuidade para um novo sistema operacional com os mesmos princípios, mas um pouco menor, posteriormente batizado de UNIX por seu colega do Bell Labs, Brian Kernighan.

Em 1973, outro pesquisador do Bell Labs, Dennis Ritchie, reescreveu todo o sistema Unix em uma linguagem de alto nível, chamada C, desenvolvida por ele mesmo. Por ter sido escrito em linguagem de alto nível, o sistema passou a ter grande aceitação por usuários externos à Bell Labs, pois oferecia uma grande portabilidade para outras plataformas de hardware.

Entre 1977 e 1981, a AT&T, alterou o Unix, fazendo algumas mudanças particulares e lançou o System III como, por exemplo, a criação de *named pipes*, a chamada de sistema ou comando *uname* e a introdução do conceito de fila de execução (*run queue*). Em 1983, após mais uma série de modificações, foi lançado o conhecido Unix System V, que passou a ser vendido. Até hoje esse sistema é usado no mercado, tornando-se o padrão internacional do Unix. O Unix é um sistema operacional proprietário pago e caro e é usado desde *workstations* até poderosos computadores (como *mainframes*) por diversas multinacionais, como a IBM em seu sistemas Z ou servidores IBM Power 795 ou supercomputadores, alguns entre os melhores 10 sistemas na lista do Top500 [3] como o IBM BlueGene/P, Fujitsu K-computer, o chinês Tianhe-1A, Cray Jaguar, etc.

[1] **Tarifação** é o modo de cômputo e cobrança em um modelo de negócio com pagamento de serviços por uso efetivo como, por exemplo, serviços de água, luz, telefone, os novos negócios como os de Computação em Nuvem, etc.

Em 1983, Richard Stallman [15], criador e defensor do *movimento do software livre*, criou o Projeto GNU [16] e mais tarde, a Fundação para o Software Livre (FSF). Um dos objetivos de Stallman era criar um sistema operacional baseado em código aberto/livre. Inicialmente, naquela data, ele criou o GNU (GNU *is Not Unix*) com seu próprio *kernel* denominado Hurd. O GNU foi uma variante de aplicações Unix compatíveis com um enorme manancial de outros aplicativos para os sistemas operacionais Unix comerciais de então (BSD, SunOS, IRIX, AIX, HP-UX, etc).

O sistema operacional MINIX [17] foi concebido como sendo Unix compatível, minimalista e barato para fins educacionais e com código livre exclusivamente para o mundo acadêmico. No GNU *Project*, grande parte do corpo do Linux foi desenvolvido, isto é, compiladores, interpretador de comando ou *shell*, comandos de linha, utilitários, etc, mas não sua essência: o *kernel*, que tomara emprestado do MINIX.

Em 1991, o jovem Linus Benedict Torvalds, estudante de Ciência de Computação da Universidade de Helsinki, Finlândia, propôs a criação de um novo *kernel* para o MINIX e foi duramente rejeitado e menosprezado pelo conservador Prof. Tanenbaum em um histórico debate [18] via troca de e-mails nos primeiros meses do ano de 1992. O debate enfatizou as convicções de cada um deles sobre o tema: sistema operacional. Linus Torvalds defendia uma arquitetura de *kernel* monolítico[2] para os novos sistemas operacionais livres, ou seja, não comerciais; já Tanenbaum, defendia um sistema operacional *microkernel*, onde os módulos do sistema seriam distribuídos por outros processos, minimizando as funções do *kernel* no "*espaço do sistema*" e maximizando-as no "*espaço do usuário*".

O Linux é um sistema operacional livre e é uma reimplementação das especificações POSIX (padronização da IEEE) para sistemas com extensões System V e BSD. Isso significa que o Linux é bem parecido com Unix, mas não vem do mesmo lugar e foi escrito de outra forma. Além disso, nada se paga por ele, diferentemente do UNIX.

Linus Torvalds desenvolveu o Linux, de maneira despojada, não objetivando ganhar dinheiro e sim criar um sistema só pelo prazer de fazê-lo [19] [20], para seu uso pessoal e que atendesse às suas necessidades. O estilo de desenvolvimento do sistema operacional que foi adotado foi o de ajuda coletiva, ou seja, os esforços coletivos da comunidade científica para a melhoria do sistema sob uma coordenação centralizada em

[2] **Um kernel monolítico** é aquele em que todos os serviços do sistema operacional são executados na *thread* do kernel principal, assim como residem na mesma área de memória.

Torvalds. Milhares de pessoas contribuíram gratuitamente com o desenvolvimento do Linux, simplesmente pelo prazer de fazer um sistema operacional melhor.

Em 25 de Agosto de 1991, Linus Torvalds postou através de mensagem para o grupo *comp.os.minix* da *Usenet* (uma de lista de discussão) que estava desenvolvendo um sistema operacional livre e que qualquer um poderia enviar contribuições como aqueles que gostavam ou não de MINIX, por exemplo. Posteriormente, em setembro de 1991, foi disponibilizada via servidor de FTP a versão do *kernel* 0.01. Depois de muitas outras versões intermediárias, foi liberada a versão 1.0.0 em 14 de março de 1994, sendo a versão 1.2.0 liberada um ano após, em março de 1995. A evolução, e as voluntárias contribuições do Linux o tornaram instalável, estável e operacional de PC's aos supercomputadores, passando por vários dispositivos embarcados e móveis do mercado como, por exemplo, *desktops*, *notebooks*, celulares, consoles de jogos (PS3, Xbox 360, Wii, Nintendo DS), *tablets*, Projeto Sequoia (IBM BlueGene/**Q**), **K** Computer (Fujitsu), etc.

Com relação ao escalonador de processos, que é o foco deste trabalho, as primeiras versões do Linux apresentavam um escalonador primordial com heurísticas bem simples. Em 1994 surge o seu sucessor, o escalonador ***O*(1)** [21] [22], objetivando principalmente maior escalabilidade[3] e incorporação de mais algoritmos de escalonamento ou, mais precisamente, políticas de escalonamento como são comumente conhecidos. Pelo mesmo motivo, logo seria substituído e atualizado pelo escalonador *Rotating Staircase Deadline (RSDL)* [23] [24]. O *RSDL* foi o seguinte escalonador do *kernel* do Linux desenvolvido pelo australiano Con Kolivas. O escalonador atualmente vigente, o inovador ***CFS*** [22] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] , foi desenvolvido pelo húngaro Ingo Molnar em 2007 [31], baseado, fundamentalmente, nas ideias de seu antecessor.

O escalonador ***CFS*** introduziu novos conceitos fundamentais para o escalonamento de processos ou tarefas em Linux como, por exemplo: uma estrutura de dados representada em árvore binária rubro-negra [36] [37] [38] e já aplicadas anteriormente no escalonador ***CFQ*** [39] ; classes de escalonamento ou conjuntos de políticas de escalonamento agrupadas por classe, por exemplo: *RoundRobin*, *RealTime*, etc; uma única fila de execução por processador tal qual o escalonador ***O(1)*** [40] [41],

[3] ***Escalabilidade*** é a característica de um sistema que indica sua habilidade de manipular uma porção crescente de trabalho de forma uniforme, ou estar preparado para crescer.

para melhorar a escalabilidade e aproveitar melhor os ambientes de multiprocessamento simétrico [42] e *multicore*, que despontavam nas novas e modernas tecnologias de processadores; um novo conceito de *virtual runtime* ou *vruntime* [30] [43] [44]; e um incipiente *Control Groups* [45] [46] [47] [48] [49] [47] concebido por desenvolvedores da IBM, o qual permite o compartilhamento de recursos pelos processos ou grupos de usuários de forma ponderada, isto é, proporcional a pesos para cada grupo definido em arquivos de configuração próprios. Esta abordagem dos Grupos de Controle possibilita a implementação de um ***QoS***, o que garante a qualidade de serviço. Além disso, este novíssimo e atual escalonador ***CFS*** passaria a integrar o próprio *kernel* do Linux a partir da versão 2.6.23 e seguintes do ano de 2007 em diante, assim como um depurador (*sched_debug*) e um gerador de estatísticas dinâmico (*schedstat*) que fazem parte do próprio escalonador, utilitários estes acessíveis ou não em tempo de geração do *kernel* ou durante a execução com a parametrização do sistema através de uma interface de controle automático de alteração dinâmica de parâmetros (*sysctl).*

Quanto mais escalável for o *kernel* e seu escalonador, isto é, quanto mais processadores puderem ser acrescentados e bem utilizados, maior será o paralelismo dos processos e das aplicações, ou seja, maior escalabilidade implica em maior paralelismo. Por escalabilidade, entenda-se: a característica de um sistema que indica sua habilidade de manipular uma porção crescente de trabalho de forma uniforme, ou estar preparado para crescer e otimizar os recursos de infra estrutura, extraindo o máximo possível das máquinas e fazendo isso com alta performance. Este paralelismo para máquinas de multiprocessamento simétrico e/ou *multicore*, significa um paralelismo em nível de *thread* do sistema e não em nível de instrução. Por isso, a escalabilidade é um pré-requisito para melhor desempenho do ponto de vista do paralelismo que se pode ou deseja obter com tecnologias SMP ou *multicore*. Em essência, é importante reduzir as latências do escalonador [50] (considerada como o tempo de tomada de decisão pelo escalonador), que é dependente da complexidade computacional dos algoritmos de tomada de decisão dos escalonadores, do número de processos a gerenciar e da proximidade física entre os processadores e as memórias, algumas destas de nível L1 e L2 que estão na própria pastilha física da CPU.

1.3 OBJETIVO

Este trabalho objetiva prever o comportamento do escalonador atual do Linux, ***CFS***, para poder aplicar metodologias ou parametrizações do escalonador visando adequá-lo para aplicações específicas, como por exemplo, as CPU intensivas dos ambientes HPC.

Através do desenvolvimento de experimentos com alteração em tempo de execução de parâmetros dinamicamente reconfiguráveis do escalonador ***CFS*** através de sua interface ***sysctl***, que é uma interface de linha de comando (**CLI** – *command line interface)*, deseja-se mostrar a previsibilidade do comportamento do sistema operacional sob vários tipos de carga: definida ou variável de processos teóricos de consumo infinito de processador ou de consumo fixo do mesmo, demonstrando-se assim a tendência de equidade de divisão do tempo de CPU para um número finito de processos. Outrossim, através de testes de configuração do módulo de Grupos de Controle ou *Control Groups*, mostramos a possibilidade de implementação, de maneira simples, de políticas de ***QoS*** com o novo *kernel* 2.6.23 ou seguintes, particularmente para tempos de processamento em sistemas SMP e *multicore*.

Estes experimentos serão realizados para serem aplicados e estendidos a outros cenários, que tenham células formadas por sistemas aqui testados como os Clusters, ambientes HPC, Nuvem Computacional, etc, nos quais possam ser aplicáveis.

1.4 ORGANIZAÇÃO DESTA DISSERTAÇÃO

Neste capítulo foram brevemente apresentados a motivação e o histórico para os escalonadores do Linux desde seu nascimento, nos primórdios de 1991, até a data atual. No 0 são descritos os escalonadores em geral e aqueles particulares que pertenceram ao *kernel* do Linux: o *naïve*, ***O(n)***, ***O(1)***, ***RSDL*** e o atual ***CFS***. No Capítulo 3 são detalhadas as novidades e funcionalidades do novo escalonador ***CFS*** do Linux, suas inovações, seu conceito de *vruntime* [44] [30], seus depurador e gerador de estatísticas, bem como seus parâmetros de escalonamento dinamicamente reconfiguráveis e/ou configuráveis em tempo de construção do *kernel*. No Capítulo 4, são apresentados os experimentos realizados em plataformas de hardware disponíveis. Os resultados experimentais são analisados e feita uma avaliação do escalonador CFS para verificar se este cumpre ao que se propõe para aplicações e/ou processos CPU-*bound ou* CPU-*intensivo*. No Capítulo 5 é introduzido o conceito de *CGroups*, a formalização de seus conceitos, seus arquivos de configuração, formas de uso, e alguns

exemplos. Finalmente, o Capítulo 6 apresenta as conclusões deste trabalho bem como, os possíveis trabalhos futuros.

CAPÍTULO 2 – ESCALONADORES DE PROCESSOS OU TAREFAS

Neste capítulo são tratados os escalonadores do Linux, todas as suas estruturas de dados fundamentais, seus objetivos e esquemas em geral, suas complexidades computacionais, políticas de escalonamento, componentes principais bem como o conceito de balanceador de carga em geral. Além disso, enfatizamos os principais parâmetros de configuração dinâmica do escalonador CFS e as formas básicas para o ajuste fino dinâmico destes parâmetros do escalonador necessários para a adaptação e adequação de aplicações com características específicas como, por exemplo, as de HPC que são CPU intensivas.

2.1 ESCALONADORES EM GERAL

O problema de escalonamento de tarefas, em sua forma geral é dito NP-completo [51] [52] [53]. A função principal de um escalonador é alocar tarefas nos processadores disponíveis da arquitetura alvo, de forma que os objetivos específicos sejam alcançados, como por exemplo, minimizar o tempo de uma dada aplicação, ou aumentar a vazão do sistema. Em linhas gerais, o escalonador deverá decidir que processo deterá o controle da CPU em um determinado instante e qual será o próximo processo a fazê-lo e quando, em um instante seguinte. Resumindo: o escalonador de processo deve decidir quem, quando, onde e durante quanto tempo um processo controlará um recurso.

Nosso escopo restringe-se aos escalonadores de processos no que tange a alocação e distribuição dos recursos computacionais (tempo de CPU, memória, rede, disco, etc) entre os processos. Algumas principais características dos escalonadores são:

- Ser justo: Todos os processos devem ser tratados igualmente, tendo possibilidades idênticas de uso do processador, devendo ser evitado o adiamento indefinido.

- Maximizar a produtividade (*throughput*): Procurar maximizar o número de tarefas processadas por unidade de tempo.

- Ser previsível: Uma tarefa ou processo deveria ser sempre executado com aproximadamente o mesmo tempo e custo computacional.

- Minimizar o tempo de resposta para usuários interativos.
- Maximizar o número possível de usuário interativos.
- Minimizar a sobrecarga (*overhead*): Recursos não devem ser desperdiçados embora algum investimento em termos de recursos para o sistema pode permitir maior eficiência.
- Favorecer processos "bem comportados": Processos que tenham comportamento previsível poderiam receber um serviço melhor.
- Balancear o uso de recursos: o escalonador deve manter todos os recursos ocupados, ou seja, processos que usam recursos subutilizados deveriam ser favorecidos.
- Exibir degradação previsível e progressiva em situações de intensa carga de trabalho.

Na prática, os objetivos acima mencionados são conflitantes como, por exemplo, tempo de resposta versus maximização de produtividade. Assim, o escalonador deve se adequar e se comprometer com os objetivos e necessidades das aplicações do usuário.

Um escalonador de processos [54] [41] [55] [56] [57] pode ser implementado, em sua essência, a partir de uma estrutura de dados associada a um conjunto de políticas ou algoritmos de escalonamento, ditas heurísticas, ou seja, para que sejam soluções ótimas ou tão boas quanto possível para todos os casos de escalonamento.

Um escalonador é de fundamental importância para o sistema operacional. Um escalonamento ruim pode causar uma distribuição injusta com mais recursos da máquina para uns processos em detrimento de outros, monopolização de recursos, por exemplo, *CPU-bound* versus *I/O-bound*, etc. No caso particular, o escalonador do *kernel* do Linux deve tomar a decisão de quem, quando e por quanto tempo um processo vai deter o controle da CPU, por exemplo, e qual será o seguinte processo a obter o controle.

O escalonador de processos para os sistemas multiprocessados modernos, deve levar em conta o balanceamento de carga, que também é um problema NP-completo [53], entre os processadores físicos e/ou critérios para a redução do consumo energético dos processadores. Isto pode ser implementado, por exemplo, concentrando-se os processos em execução por afinidade dos processadores envolvidos (em uma mesma pastilha ou pacote físico), evitando-se assim, sempre que possível, uma

distribuição esparsa de execução dentre os processadores, ou seja, processadores ociosos misturados com ativos em um processador físico.

2.2 O ESCALONADOR PRIMORDIAL

O primeiro escalonador do Linux 1.0.x [51] [58] era tecnicamente muito simples. Consistia de uma estrutura formada por uma lista circular duplamente encadeada na qual cada processo era representado em um nó e operava com uma política de escalonamento do tipo *round-robin*. A pesquisa do escalonador era feita sequencialmente na lista circular e, portanto, tinha uma complexidade de $\boldsymbol{O(n)}$.

O escalonamento baseava-se em *threads*. Todo processo obtinha um número fixo de *ticks*, por default 15. Considerando que o relógio executava a 100 Hz, cada tique equivalia a 10 ms, denominado instante. A prioridade do processo era dada em *ticks* e também era denominada *quantum*. A cada interrupção do relógio, o contador do processo corrente era decrementado, e quando o contador chegasse a zero, a *flag* "*need_resched*" era ativada. O escalonador verificava todos os processos e selecionava o de maior prioridade, ou seja, aquele que tinha o maior número de *ticks*, e deixava-o executar. Havia também o conceito de época, que era o período no qual todos os processos executavam durante as suas respectivas fatias de tempo. As fatias de tempo eram calculadas no início de cada época. Se todos os processos tivessem zero *ticks* restantes, o escalonador realizava o algoritmo do envelhecimento (*aging algorithm*) e tentava novamente. A função "*wake-up()*" também ativava a *flag* "need_sched" se a tarefa "despertada" ou "pronta para executar" tivesse o contador de *ticks* maior que o da tarefa corrente em execução.

Para aquela época o escalonador era suficiente pois o Linux também era somente console. O desempenho era de até 6 (seis) processos por segundo e se um deles fosse de uma interface gráfica, o usuário teria que esperar quase um segundo para ver o mouse se mexer, por exemplo. Com a evolução do hardware e das aplicações (i.e. como as interfaces gráficas), o escalonador teve que ser melhorado.

Dada sua simplicidade, o escalonador não era adequado para máquinas SMP, pois, a cada tomada de decisão por parte do escalonador, a única e geral fila de processos era bloqueada para a pesquisa sequencial. Era simples e rápido ao que se destinava naquela época, porém, devido à evolução tecnológica do hardware, foi rapidamente substituído. O escalonador seguinte da versão Linux 2.2 introduziu a ideia de classes de escalonamento, permitindo políticas de escalonamento para tarefas de

tempo real ou não. Esta última versão também incluiu suporte a arquiteturas de multiprocessamento simétrico.

2.3 O ESCALONADOR *O*(*N*)

Incluso na versão Linux 2.4 [17] [59] [60], o escalonador denominado ***O*(*n*)** operava em uma complexidade de tempo da ordem de ***O*(*n*)**. Esquematicamente, o escalonador dividia o tempo em épocas, e dentro de cada época era permitido, a toda tarefa executar até que sua fatia de tempo expirasse. Caso não executasse toda a sua fatia de tempo de CPU devido a uma interrupção ou requisição de I/O por parte da tarefa, por exemplo, a metade do tempo restante era adicionada à nova fatia de tempo para permitir que a tarefa executasse durante mais tempo na próxima época. O escalonador deveria simplesmente interagir com as tarefas aplicando uma métrica denominada função bondade (*goodness*) para determinar qual seria a próxima tarefa a ser executada. Ela dependia da prioridade e do quantum do processo através das seguintes regras:

```
if (classe  ==  tempo_real)
      bondade = 1000 + prioridade;
if (classe  ==  tempo_compartilhado && quantum > 0)
      bondade = quantum + prioridade;
if (classe  ==  tempo_compartilhado && quantum = 0)
      bondade = 0;
```

Constatou-se que a função bondade privilegiava as tarefas de tempo real e aquelas cujo *quantum* ainda não tinha sido esgotados. De outra forma, nenhuma bondade era concedida.

Apesar da simplicidade, o escalonador ***O*(*n*)** era ineficiente e carecia de escalabilidade, além de sua fraqueza para sistemas de tempo real, pois os eventos de tempo real não podiam ser tratados com a urgência necessária exigida com a qual ocorriam. Também faltavam recursos do escalonador para explorar as novas arquiteturas de hardware como a de processadores com múltiplos núcleos.

Sobre a implementação de uma fila de execução para cada CPU física ou lógica, pode-se dizer que foi criada para resolver os problemas referentes a saltos (desnecessários e custosos) de uma CPU para outra, dados pelos processos em um efeito *ping-pong*. Uma fila por processador também minimizava a contenção da escolha e

definição de que CPU executaria que processo. Naquele momento, esta solução era ideal para as máquinas SMP e hoje, melhor ainda para as máquinas *multicore*. Pode-se visualizar na Figura 2-1, a representação da mudança de uma fila de execução global única para múltiplas filas de execução por CPU.

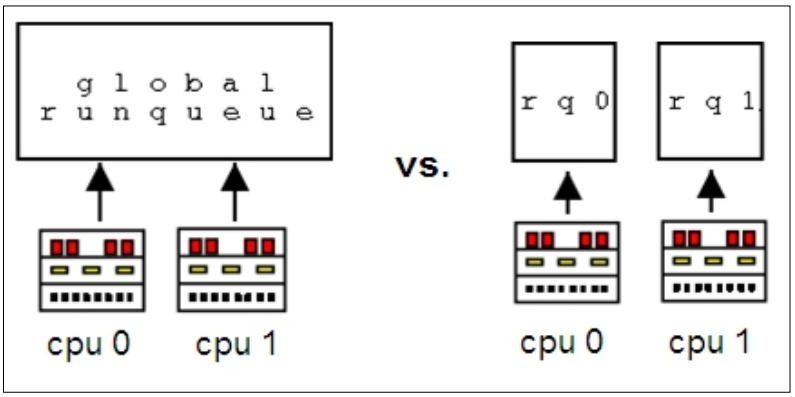

Figura 2-1 Diagrama da fila de execução global × fila de execução única por processador

A estrutura de dados do escalonador ***O(n)*** pode ser vista na Figura 2-2, isto é, era representada por uma lista circular duplamente encadeada cuja complexidade de inserção/exclusão de um processo poderia ser realizada em um tempo ***O*(*1*)** mas a busca em um tempo ***O*(*n*)**, que deu o nome ao escalonador.

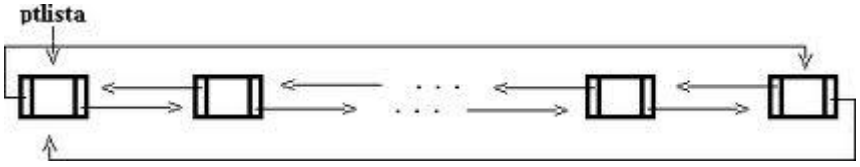

Figura 2-2 Estrutura da fila de processos como lista circular duplamente encadeada.

2.4 O ESCALONADOR *O*(*1*)

O escalonador inicial da versão Linux 2.6 [61] [62] foi projetado para resolver muitos dos problemas de seu antecessor e não necessitava percorrer toda a lista de tarefas para identificar a próxima a ser escalonada, ou seja, era muito mais eficiente e

escalável que o anterior com uma complexidade de tempo constante ou como passou a ser denominado de ***O(1)***.

O escalonador ***O(1)*** mantém o controle das tarefas executáveis em uma fila de execução (na verdade, em duas filas de elementos compostos para cada nível de prioridade – uma para as tarefas ativas e outra para as tarefas expiradas [63]) criada e mantida para cada CPU existente do sistema. Isto significa que para identificar a próxima tarefa a executar, o escalonador simplesmente precisava retirar a identificação do processo da fila de execução ativa específica por prioridade[46]. As tarefas de mesma prioridade são alternadas em *round-robin*. Cada processo recebe uma fatia de tempo de CPU de acordo com sua prioridade, inicialmente zero, e que pode ser modificada via chamada *nice()* do sistema. Quando um processo é criado, ele é movido para a lista ativa. Depois que um processo interativo termina de executar, durante toda a sua fatia de tempo, ele é movido para a fila de expirados. Um processo interativo só é movido para a fila de expirados em dois casos: se o processo que estiver há mais tempo na fila de expirados tiver esperado um tempo limite máximo ou se existir um processo na fila de expirados cuja prioridade estática seja maior que a do processo interativo. Desta forma, o escalonador dá prioridade aos processos interativos e previne a inanição (*starvation*). Os processos da fila de expirados só serão executados quando todos os elementos da lista ativa tiverem sido executados, ou seja, a lista de ativos estiver vazia. A principal estrutura é responsável pelo tempo ***O(1)*** do algoritmo.

O escalonador ***O(1)*** era mais escalável e incorporava mais métricas de interatividade com numerosas heurísticas para determinar se tarefas eram voltadas para leitura/gravação ou cálculos intensivos, ou seja, *I/O-bound* ou *CPU-bound*. No entanto, o escalonador ***O(1)*** torna-se ineficiente no *kernel* por necessitar de duas estruturas que se replicavam e que precisavam de gerenciamento. A grande porção de código necessária para o cálculo de heurísticas para a tomada de decisão do próximo processo a ser executado foi fundamentalmente difícil de gerenciar, implicando em uma grande latência do escalonador, isto é, um custo alto para o escalonador tomar sua decisão.

A Figura 2-3, mostra a estrutura de dados composta por duas listas de prioridades para cada CPU, uma com os processos em execução, aquela denominada ativa e a outra com os processos que já esgotaram a sua fatia de CPU, a dita expirada.

As prioridades de 1 a 100 são reservadas para os processos que necessitam de execução em tempo real; já as outras prioridades de 101-140 são reservadas para os processos comuns ou interativos.

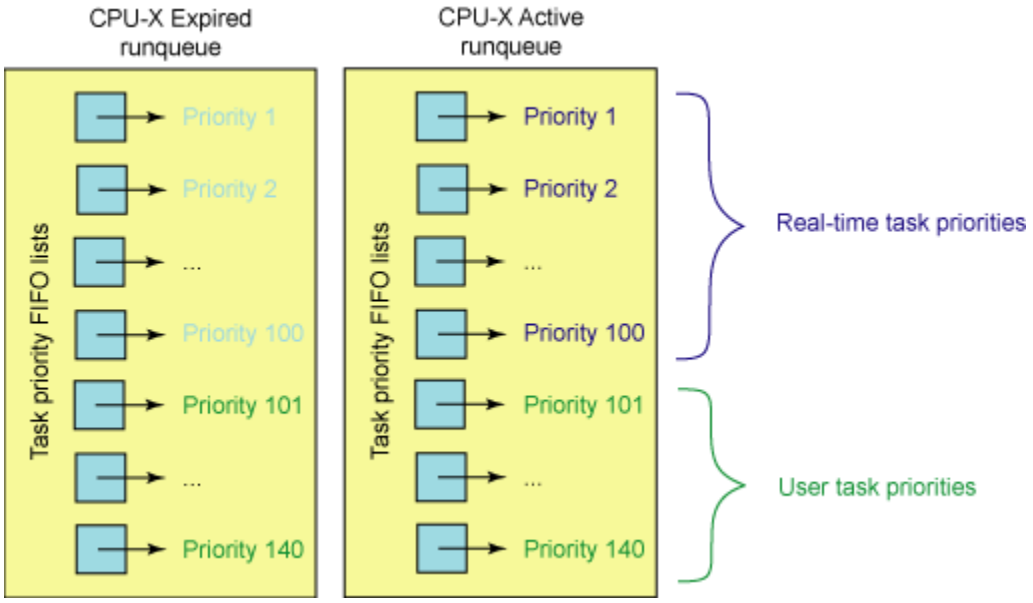

Figura 2-3 Estrutura básica da fila de execução de processos como duas listas ativa e expirada indexadas por prioridades [26].

Eram necessárias algumas mudanças para resolver os problemas de desempenho das aplicações interativas apresentados pelo escalonador ***O(1)***. A mudança veio na forma de um *patch* de *kernel* de seu criador e desenvolvedor Con Kolivas com o seu escalonador *Rotating Staircase Deadline*. O resultado foi um escalonador com projeto mais simples e que incorporava uma melhor equidade com baixa latência de escalonamento. O escalonador de Con Kolivas impressionou a muitos e foi incorporado ao *kernel* 2.6.21. Esse escalonador será descrito a seguir.

2.5 O ESCALONADOR RSDL → SD

O escalonador ***RSDL*** [23] [24], posteriormente denominado ***SD***, pretendia aumentar o posicionamento do Linux no mercado de *workstations* e *desktops*, melhorando o tempo de resposta experimentado pelo usuário assim como o sentimento de interatividade com o mesmo.

O escalonador ***SD*** mantém um *array* de prioridades mostrado na Figura 2-4. Em cada nível existe uma lista de processos que possuem uma quota de tempo concedida para executar naquela prioridade. Aos processos de maior prioridade são dadas fatias de tempo de execução, então o escalonador rotaciona os processos de diferentes prioridades usando um algoritmo do tipo *round-robin*.

Quando um processo usa sua quota em um dado nível de prioridade, ele cai para a próxima prioridade e recebe uma nova quota. Todos os processos em um nível passam para um nível seguinte de prioridade mais baixa independentemente de terem consumido sua cota de CPU ou depois que cada nível de prioridade consome sua própria cota. Os processos podem continuar sua execução, mas somente depois que os processos de maior prioridade tiverem tido a sua vez. Os processos movem-se como se descessem uma escada. Como resultado deste mecanismo de "pequena rotação", os processos que esperam em níveis de prioridade mais baixa precisam apenas aguardar um período limitado de tempo antes de todos os outros processos que estão sendo executados em seu nível. A latência máxima para qualquer processo em espera para execução é, assim, limitada, e pode ser calculada, não hvendo inanição (*starvation*) com este escalonador. Caso cheguem mais e mais processos com maior prioridade estes serão alocados em suas específicas filas de prioridade, executarão a parte que lhes cabe e descerão a escada naturalmente, perdendo prioridade sem necessariamente monopolizar as filas de prioridades mais altas, e permitindo que outros processos anteriores possam consumir suas cotas de CPU.

À medida que os processos usam o seu tempo, eles são movidos para um segundo array, chamado *array* de processos "expirados". Nele, os processos são colocados de volta em sua prioridade original. Processos no *array* de expirados não são executados, eles são deixados de fora, como se fossem congelados, até que não existam mais processos no *array* de processo “ativos”, ou até que todos os processos saim do *array* de processos ativos como um resultado das rotações menores. Nesse ponto, uma "rotação principal" acontece: os *arrays* de “ativos” e “expirados” são trocados em uma única operação (**rq** -> *active* = **rq** -> *expired*;), portanto em um tempo ***O(1)***, e toda a série de eventos é reiniciada a partir da nova fila de “ativos”. A Figura 2-4 apresenta, simplificadamente, o esquema correspondente aos *arrays* do escalonador SD.

O RSDL, diferentemente do escalonador ***O(1)***, eliminou o código de estimativa de interatividade, que era um algoritmo baseado em estatísticas para tentar prever o futuro (e falhou nisso); é baseado no conceito de justiça estrita, onde todos os processos são tratados igualmente e recebem o mesmo tempo de processamento; melhorou a “interatividade” percebida pelo usuário; usa o conceito de filas de prioridade ativa e expirada, portanto, é afetado pelo conceito de troca de array expirado para ativo da mesma forma que o ***O(1)***.

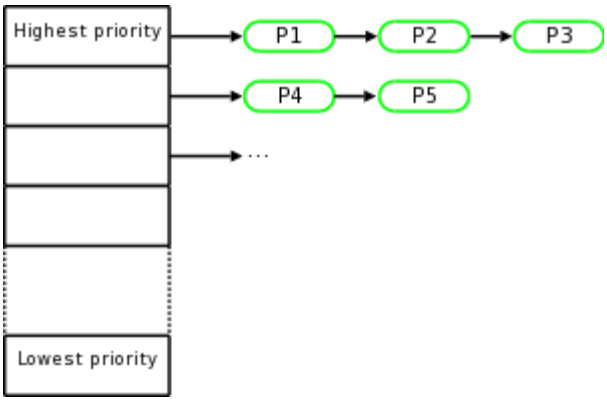

Figura 2-4 Diagrama simplificado do escalonador SD [23]

2.6 ESCALONADOR CFS

O escalonador ***CFS*** (*Completely Fair Scheduler*) [32] [64] [65] desenvolvido pelo húngaro Ingo Molnar, foi incorporado ao *kernel* do Linux na versão 2.6.23, e liberado em julho de 2007, por decisão de Linus Torvalds [32] em detrimento do escalonador ***SD*** de Con Kolivas criado em 2004 [66]. O objetivo principal do ***CFS*** era o de ser um escalonador completamente equânime, na verdade, tanto quanto possível. Segundo o seu desenvolvedor, Ingo Molnar, "80% do projeto do ***CFS*** poder ser resumido em uma única frase: o ***CFS*** basicamente modela '*uma CPU ideal, multitarefa e precisa*' em um *hardware* real." [31]. Por motivos de necessidade de desempenho, o ***CFS*** visa resolver os problemas de melhoria de tempo de resposta e a sensação de interatividade experimentada pelos usuários do Linux em *workstations e desktops*, principalmente. Além desses, *notebooks* e outros dispositivos móveis disponíveis no mercado com sistemas pré-instalados ou embarcados.

Como sabemos, a essência da implementação de um escalonador reside em como este representa sua estrutura de dados para armazenamento das informações relevantes sobre os processos e o seu conjunto de políticas de escalonamento propriamente dito, baseado em heurísticas com objetivos definidos: FCFS, SJF, PRI, RR, etc (estas são heurísticas ou políticas clássicas de escalonamento empregadas em outros escalonadores). O ***CFS*** inova com suas filas de execução representadas em suas estruturas de dados formadas por árvores binárias rubro-negras (*rb-tree*) [37] [38] como exemplificado na Figura 2-5; sem HZ ou *jiffies*; não utilização de tempo de *sleep*; não

identificação de processos interativos; ausência de array de prioridades; e por sua heurística simples.

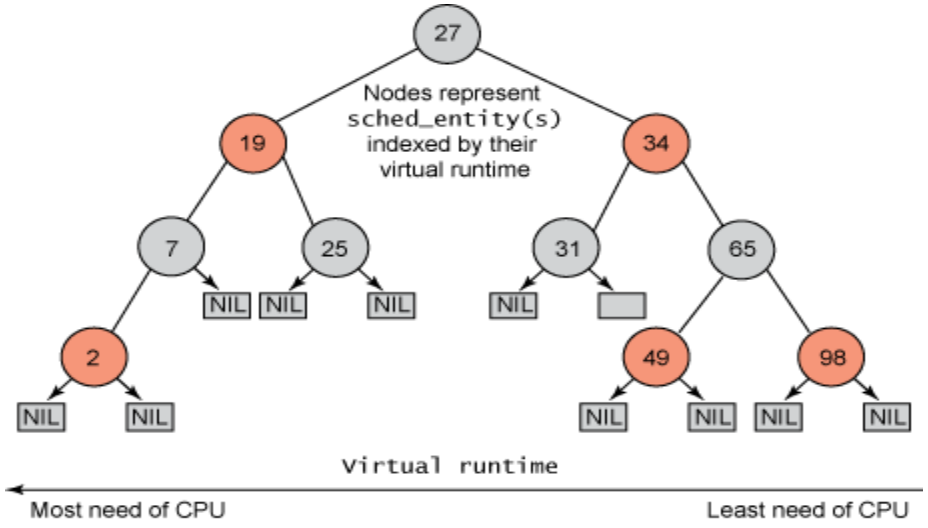

Figura 2-5 Estrutura da fila de execução em árvore rubro-negra.

Convém lembrar, HZ é taxa de frequência de demarcações ou *ticks* do temporizador ou relógio do sistema por segundo; e *jiffy* é 1/HZ, o inverso de HZ, isto é, o tempo transcorrido entre dois *ticks* do temporizador do sistema.

O ***CFS*** tomou como exemplo o ***CFQ*** (escalonador de filas de Entrada/Saída) e o *Rotating Staircase Deadline (**RDSL**)*, com referência à estrutura de dados do primeiro e a forma de implementação de políticas do segundo. Mas, de fato, o ***CFS*** tem personalidade própria e também, com as melhorias introduzidas no *kernel* 2.6.24 passa a ter um controlador de grupos de tarefas *CGroups* ou *ControlGroups*, permitindo assim uma possibilidade de ***QoS*** *(Quality of Service)* e garantias de ***SLA***s (*Service Level Agreements*). Os pilares do ***CFS*** fundamentam-se em:

- temporizadores de alta-resolução [67] [68] [69] da nova biblioteca ***hrtimer*** do Linux com tempos de precisão de nanossegundos (ns) no lugar dos de milissegundos (*ms*), introduzidos no *kernel* 2.6.16 [70]. Isto possibilitou, entre outras coisas, o aumento da resolução para eventos de escalonamento;
- filas de processos de execução representadas por árvores rubro-negras, uma para cada processador lógico (*core* físico ou via SMT, isto é, *threads* habilitadas por *core*); e,

- uma divisão aritmética do tempo de processamento de modo igualitário para todos os processos em execução, equivalente ao descrito no algoritmo de compartilhamento proporcional de recursos [31] [40] [71] [72]. Por isso, o escalonador ***CFS*** é considerado "livre" de heurísticas, isto é, o tempo de CPU é dividido igualmente por todos os processos de uma mesma fila de CPU e por todos os processos de todas as filas de execução do sistema. Portanto, não existem complexos algoritmos de decisão para o próximo processo a executar, salvo o simples de compartilhamento proporcional ao número de processos.

Os tempos medidos em nanossegundos para os principais parâmetros de escalonamento (*kernel_sched_latency*, *kernel_sched_min_granularity* e *kernel_sched_min_wakeup_granularity*) permitem que sejam disponibilizadas frações de tempo de CPU menores, para que estas sejam distribuídas mais frequentemente para os processos em execução. Isto necessariamente aumenta o número de trocas de contexto, mas implica em maior justiça na distribuição do tempo de CPU como um todo. Logo, favorece os processos interativos e a subjetiva melhoria de desempenho experimentada pelo usuário.

2.6.1 Principais componentes do escalonador *CFS*:

Esquematicamente, o escalonador ***CFS*** pode ser visualizado de forma sucinta na Figura 2-6. A seguir, suas funcionalidades e os seus componentes principais são descritos.

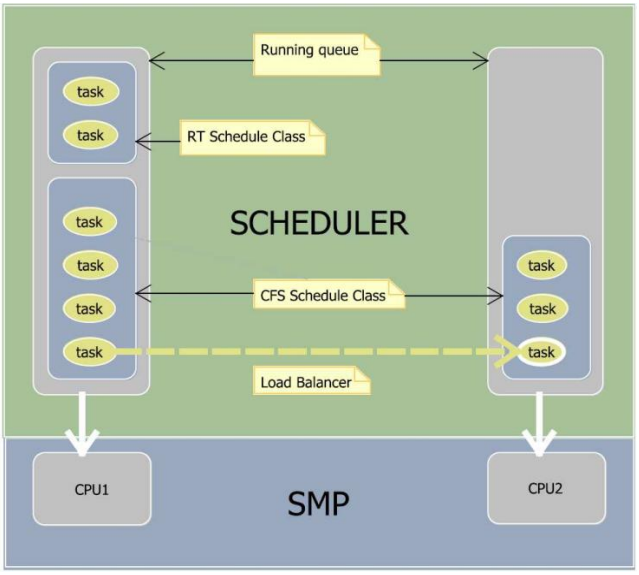

Figura 2-6 Diagrama geral do escalonador *CFS* [26]

O escalonador ***CFS*** gerencia através de suas classes de escalonamento **RT**, para os processos de tempo real com as políticas: *SCHED_FIFO, SCHED_RR*, e ***CFS*** para os processos tratados pelo algoritmo ***CFS*** para os processos comuns dos usuários com as políticas: *SCHED_NORMAL (SCHED_OTHER), SCHED_BATCH, SCHED_IDLE,* as filas de execução, uma para cada CPU, decidindo que processo de determinada fila terá em que instante e durante quanto tempo o controle da CPU daquela fila. Os processos de tempo real (**RT)** possuem uma fila própria para cada CPU e não são tratados pelo algoritmo **CFS** do escalonador ***CFS***. Os processos de tempo real são classificados pelo *kernel* com prioridades de 0 a 99 e são tratados pelo escalonador ***CFS*** antes de todos os outros ditos processos comuns, cujas prioridades variam de 100 a 140. De tempos em tempos, o componente denominado balanceador de carga procura equilibrar a distribuição de processos nas filas de execução, visando distribuir uniformemente a carga de processos do sistema entre as CPUs. Os principais componentes são os seguintes:

Fila de Execução: uma fila de execução (*running queue* – **rq**) é criada para cada processador (**CPU** ou ***core***), uma para os processos comuns (***cfs_rq***) e outra para os processos de tempo real (***rt_rq***), cada uma delas com sua estrutura correspondente. É definida no *kernel/sched.c* como *struct_runqueue*. Cada ***rq*** contém uma lista de processos executáveis em um dado processador. Uma *struct_runqueue* é definida em *sched.c*.

Classe de Escalonamento: o conceito de classe de escalonamento foi introduzido no *kernel* 2.6.23. É uma hierarquia extensível dos módulos do escalonador. Os módulos encapsulam detalhes das políticas de escalonamento e são chamados a partir do núcleo do escalonador sem que o núcleo assuma muito mais sobre elas. As classes de escalonamento são implementadas através das *sched_class structure*, que contêm ganchos para as funções que devem ser chamadas sempre que ocorram eventos relevantes como, por exemplo, a criação de um processo responsável por decodificar um filme ativa funções com mecanismos de tempo real ou a criação de uma sessão de terminal ativa funções da classe ***CFS***, que trata processos interativos. Os processos referenciam-se às políticas de escalonamento através das *struct task_.sched_class*. Existem duas classes principais implementadas no *kernel* 2.6.32:

1. Classe de Escalonamento Completamente Equânime (*fair class*): escalona os processos que seguem o algoritmo do Escalonador Completamente Equânime (***CFS***); As tarefas com prioridades de 100 a 140 têm políticas configuradas para *SCHED_NORMAL (SCHED_OTHER), SCHED_BATCH, SCHED_IDLE* e são escalonadas por esta classe de escalonamento. As implementações dessas classes estão em *kernel/sched_fair.c.* Está classe é denominada justa (*fair).*

2. Classe de Escalonamento **RT**: escalona os processos que seguem o mecanismo de **tempo-real** definido no padrão POSIX. As tarefas com prioridades de 0 a 99 têm políticas configuradas para *SCHED_FIFO, SCHED_RR* e são escalonadas usando-se estas classes de escalonamento. As implementações desta classe estão em *kernel/sched_rt.c*.

Balanceador de carga: A função do balanceador de carga é distribuir uniformente a carga de processamento das tarefas dentre os processadores ou CPUs existentes no sistema. O objetivo é preservar, tanto quanto possível, uma carga de processos igualitária entre os nós de processamento, a fim de otimizar a utilização de recursos, maximizar o desempenho, minimizar o tempo de resposta e evitar sobrecarga, impedindo assim que a carga de processamento se concentre em algumas ou poucas CPUs. Nos atuais ambientes multiprocessados, cada CPU tem sua própria fila de execução, ***rq***. Estas filas tendem a ficar desbalanceadas com o passar do tempo, isto é,

com quantidades diferentes e desproporcionais de processos por fila. Uma fila de execução sem processos leva sua CPU associada ao estado ocioso, o que implica em não se beneficiar de todas as vantagens dos sistemas multiprocessados simétricos (SMP). O **balanceador de carga** se destina a estes casos. Ele é invocado em intervalos definidos ou todas as vezes que o escalonador do *kernel* identifica filas vazias para a atuação da função de balanceamento. Caso as filas de execução estejam desbalanceadas, isto é, as quantidades de processos por fila de execução não sejam uniformemente distribuídas, o balanceador de carga tentará redirecionar as tarefas em execução nos processadores mais ocupados ou com mais processos na sua fila, para os processadores mais ociosos ou com menos processos em sua fila.

2.6.2 Ajuste fino dos principais parâmetros do escalonador *CFS*

O escalonador ***CFS*** oferece um número de parâmetros que permitem um ajuste fino de seu comportamento às suas necessidades. Estes parâmetros podem ser acessados via o pseudo-sistema de arquivos ***/proc***, isto é, um sistema de arquivos especial que existe apenas em memória e forma a interface entre o *kernel* e o espaço de usuário. O conteúdo de qualquer um de seus arquivos é gerado quando uma operação de leitura é efetuada. O *kernel* usa os arquivos em ***/proc*** para exportar seus dados internos para as aplicações do espaço do usuário e para modificar parâmetros do próprio *kernel*, incluindo os parâmetros do ***CFS***. Isto pode ser feito pelo administrador do sistema fazendo uso da interface de comando de linha ***sysctl*** ou modificando diretamente os arquivos de configuração dos parâmetros como arquivos de texto comum que podem ser lidos ou escritos, como será visto a seguir. O *kernel* se modifica dinamicamente para a nova configuração estabelecida com a alteração dos parâmetros do escalonador. Os parâmetros configurados podem ser exibidos ou modificados. Assim, para se obter o conteúdo do arquivo ***/proc/sys/kernel/sched_latency_ns*** é necessário usar um dos seguintes comandos:

```
# cat      /proc/sys/kernel/kernel.sched_latency_ns
# sysctl   sched_latency_ns
```

Para modificar seu conteúdo podemos usar uma ou outra das seguintes opções:

```
# echo     valor > /proc/sys/kernel/sched_latency_ns
```

```
# sysctl  -w  sched_latency_ns = valor
```

Onde *valor* deve ser fornecido em unidades de nanossegundos (*ns*) para os parâmetros que terminam explicitamente em *_ns*.

2.6.3 Alteração em tempo de execução dos parâmetros dinamicamente reconfiguráveis do escalonador CFS

Esta seção apresenta brevemente alguns dos parâmetros do escalonador ***CFS,*** que são dinamicamente reconfiguráveis. Eles passam a valer a partir da alteração via comando ***sysctl***, como descrito no parágrafo 2.6.2 anterior, e modificam o escalonador do *kernel*. Todos podem ser encontrados no diretório ***/proc/sys/kernel*** em arquivos texto com os mesmos nomes destes parâmetros.

sched_latency_ns - É período de tempo durante o qual todos os processos deveriam usar a CPU (por exemplo, 20000000 ns = 20 ms por *default* para *workstation*)

sched_min_granularity_ns – É o período de tempo mínimo que um processo deve ter de CPU (por exemplo, 4000000 ns = 4 ms por *default* para *workstation*)

O escalonador ***CFS*** verifica se a seguinte desigualdade é verdadeira:

$$nr_running > \frac{sched_latency_ns}{sched_min_granularity_ns}$$

onde ***nr_running*** é igual ao número de processos na fila de execução. Se esta relação é satisfeita, então o escalonador percebe que existem muito mais processos a serem executados no sistema do que a latência do escalonador permite. Então, é necessário estender a latência do escalonador para que todos os processos possam ser executados durante o período de granularidade mínima pelo menos uma vez durante o período de latência estendido. Isto é feito de acordo com o produto abaixo:

$$periodo = sched_min_granularity_ns \text{ x } nr_running$$

Isto é, o período é igual a granularidade mínima multiplicada pelo número de processos em execução da classe *fair*. Isto significa que para os valores *default* de parâmetros do escalonador, a latência de execução deve ser aumentada quando o

número de tarefas é maior do que cinco (*default* escolhido por opção e conveniência do desenvolvedor, que iniciou seus testes em uma máquina *quadcore*).

sched_child_first - este parâmetro influencia a decisão por parte do escalonador da ordem de execução do processo pai e de seu filho. Se é igual a 1 (um - valor *default*), então o processo filho ganhará CPU antes de seu pai. Se o parâmetro é 0 (zero) – o inverso acontecerá.

sched_compat_yield - este parâmetro é usado para bloquear o processo corrente de ceder voluntariamente **CPU** configurando seu valor para zero (*default*). Um processo obrigatoriamente deveria ceder a **CPU** pela chamada da função *sched_yield()* pelo escalonador mesmo que o tempo de CPU correspondente a sua granularidade mínima não tenha sido esgotado.

sched_wakeup_granularity_ns - este parâmetro define uma concessão adicional; é permitido a um processo usar a CPU por esta quantidade de tempo, mesmo que tenha exaurido sua permissão primária de CPU descrita pela granularidade mínima. Quanto maior o valor, mais difícil será para a tarefa que detém o controle da CPU sofrer a preempção. Um estudo sob suas implicações poder ser visto em [56].

sched_migration_cost - este parâmetro é usado pelo escalonador para determinar se o procedimento para selecionar o próximo processo deve ser chamado quando um outro processo está pronto para execução. Se o tempo de CPU para o processo corrente e o de um processo pronto para execução são menores do que este parâmetro (*500000 ns = 0.5 ms*, por *default*), o escalonador escolhe outra tarefa. Para que isto funcione a componente *WAKEUP_OVERLAP* deve estar ativa.

No caso de sistemas computacionais com múltiplos processadores, o escalonador também tenta espalhar a carga pelas CPUs disponíveis. A esse respeito, o escalonador usa os seguintes parâmetros:

sched_migration_cost - o custo da migração de tarefas entre CPUs (0.5 ms por *default*) que é usado para estimar se o código de um processo ainda está presente na *cache* da CPU. Se o tempo de CPU de um processo é menor do que o valor deste

parâmetro então o escalonador assume que ele ainda está na *cache* e tenta evitar mover o processo para outra CPU durante o procedimento de balanceamento de carga.

sched_nr_migrate - o número máximo de processos que o escalonador manipula durante o procedimento de balanceamento de carga. (32 por *default*).

O escalonador ***CFS*** torna possível o controle do tempo de CPU de processos de tempo real até a resolução de microssegundo.

sched_rt_runtime_ns - o tempo máximo de ***CPU*** que pode ser usado por todos os processos de tempo real (1 segundo, por *default*). Quando esta quantidade de tempo é usada, estas tarefas devem esperar por um período definido por ***sched_rt_period_us*** antes que seja permitida uma nova execução.

sched_rt_period_us - o escalonador ***CFS*** espera esta quantidade de tempo (0.95s, por *default*) antes de escalonar qualquer processo de tempo real novamente.

sched_features - representa certos componentes do escalonador que podem ser ligados ou desligados durante a operação do sistema.

Observações: Os parâmetros acima descritos são fundamentais para a reparametrização dinâmica do escalonador ***CFS***, através da interface de comandos de linha via ***sysctl***. Mudanças nesses parâmetros provocam alterações significativas na tomada de decisão do escalonador, modificando-a segundo os objetivos do administrador do sistema, visando adaptar o escalonador ***CFS*** para determinadas características das aplicações. No caso RT, os processos sofrem preempção por outros processos RT de maior prioridade e causam preempção em qualquer processo comum, isto é, da classe equânime (*fair class*).

Este trabalho dá ênfase, a modificações dos parâmetros ***sched_latency_ns, sched_min_granularity_ns*** e ***sched_wakeup_granularity_ns***, pois estes parâmetros podem causar significativas alterações no comportamento do escalonador ***CFS***, favorecendo processos CPU intensivos, que são importantes para aplicações HPC, por exemplo.

2.6.4 Determinação do *time slice* para cada processo no *CFS*

Na da verdade, o ***CFS*** não possui o conceito de *time slice* fixo. O ***CFS*** calcula dinamicamente uma granularidade mínima de CPU equivalente para cada processo executável. Esta granularidade mínima dinâmica é calculada como segue, onde:

N_T	é o número de processos na fila de execução,
$sched_latency_ns$	é o período mínimo, época mínima ou latência do escalonador, e
$sched_min_granularity_ns$	é o tempo da granularidade mínima.

$$MinTasks = \frac{sched_latency_ns}{sched_min_granularity_ns}$$

Period é tempo durante o qual todas as N_T tarefas da fila de execução executaram pelo menos uma vez.

$$Period = sched_latency_ns, se\ N_T \leq MinTasks$$

$$ou$$

$$Period = sched_min_granularity_ns\ x\ N_T, se\ N_T > MinTasks$$

$$TimeSlice = \frac{Period}{N_T}$$

Na versão 2.6.32 do *kernel* do Linux para *workstation*, por *default* a granularidade mínima é de *4000000 ns = 4 ms* e todas as tarefas em uma fila de execução tem o mesmo peso ou prioridade. O cálculo do *time slice* e a decisão de escalonamento são tomados independentemente de qual é a fila de execução (uma fila de execução por CPU física ou lógica (SMT ou HT).

2.7 TRABALHOS CORRELATOS

A tarefa de testar escalonadores no sistema operacional Linux é desafiante.

O primeiro trabalho, "*Testing Task Schedulers on Linux System*" [73], busca criar todo um *framework* de testes de escalonadores para os sistemas Linux em geral. Inicialmente ele estabelece os dois problemas fundamentais: a identificação das propriedades do escalonador a ser testado e como fazê-lo, isto é, que APIs (*Application Programming Interface*) são suficientemente precisas e portáveis para as diversas plataformas existentes. Comenta a mudança do escalonador ***O(1)*** para ***CFS*** e resolve aplicar sua metodologia apenas ao mais novo escalonador, ***CFS***. Afirmando a dificuldade formal de aplicar os programas de benchmark disponíveis aos escalonadores atuais, por considerar que estes *benchmarks* levam os escalonadores para ambientes de operação e teste atípicos e aplicáveis a apenas algumas propriedades dos escalonadores. Logo, estabelece a seguir, uma abordagem mais sistemática. Busca, teoricamente, identificar as propriedades mais importantes: a eficiência do escalonador, o seu comportamento sob carga pesada e o escalonamento de tarefas interativas.

São identificados cinco testes de caráter geral para sistematização: teste de eficiência, teste de interatividade, teste sob diferentes cargas, teste de escalabilidade e testes de equidade. A eficiência, definida no contexto do escalonador, representa a habilidade do sistema de efetuar mais operações, isto é, completar mais tarefas em menos tempo usando os mesmos recursos do sistema. Exemplifica, teoricamente, com uma coleção de Jobs sob um o mesmo escalonador que complete em menos tempo que uma segunda, caracterizando a primeira como mais eficiente.

Para os testes de interatividade, considera-se o tempo de resposta para comandos do usuário, eventos de entrada/saída, aplicações interativas, aplicações multimídia ou similares. Todas estas aplicações possuem a mesma característica: não necessitam de muito processamento mas, quando precisam, devem ser atendidos tão rápido quanto possível. Quanto ao teste de diferentes cargas, estes correspondem a simulação de múltiplas sessões de terminal, execução de aplicações multimídia com som e/ou imagem e comandos de linha emitidos pelos usuários, este deve permitir testar o comportamento do escalonador sob diferentes cargas do sistema que envolvam mudanças de carga e repetições de testes.

Identificando a tendência e presença em quase todos os novos sistemas de computação de tecnologias multiprocessadas como multiprocessadores reais ou

multicore, o teste de escalabilidade verifica a importância de inserir o comportamento do escalonador nesses ambientes multiprocessados. Os testes de escalabilidade dos escalonadores não requerem nenhuma mudança especial nos testes do *framework* mas apenas uma adequação dos parâmetros destes e a apropriada interpretação dos resultados. O escalonador deve se certificar das características do *hardware* e tentar explorá-las para obter o desempenho máximo ou ótimo.

Nos sistemas *single-user*[4], para os testes de equidade, os sistemas devem satisfazer os requerimentos de um único usuário. As aplicações usados por um único usuário são mais importantes para ele em cada instante e devem receber a máxima atenção do sistema para satisfazê-lo. O escalonador deveria certificar-se das aplicações *online* e priorizá-las quando identificar que estas tornaram-se ativas. Os sistemas multiusuário devem ser equânimes, ou seja, devem procurar dividir os recursos computacionais de maneira igualitária dentre todos os usuários do sistema. A equidade pode ser definida por tarefa ou por usuário. No conceito por usuário, o escalonador justo equilibra diferentes tempos de execução de tarefas de usuários tal que eles distribuam tempos de computação disponíveis igualitariamente. No conceito por tarefa, todas as tarefas devem obter quantidade de tempo de computação justa (equidade). Entretanto nem todos os usuários e nem todas as tarefas são iguais. Algumas são mais importantes dos que outras (por exemplo, o usuário "*root*", o administrador do sistema, comparado ao usuário normal).

Os tempos de processamento devem ser mensurados com precisas chamadas de sistemas, tais como *clock_gettime()*. Pausas ou retardos podem ser implementados por funções de *sleep* do tipo *clock_nanosleep()*, escolhidas devido a sua precisão de nanossegundo. Outras considerações detalhadas de medição de tempo de execução são consideradas no trabalho mas não são relevantes no nosso contexto de avaliação.

Os testes referentes ao primeiro trabalho correlato foram implementados em uma única distribuição Ubuntu com o kernel 2.6.31 em plataforma *x86_64* de um a quatro *cores* com a frequência do processador configurada para o máximo e o *hyper-threading* desabilitado.

Os resultados experimentais são muito simplificados mas significativos mostrando um excelente compatibilidade com os valores teóricos não explicitados mas

[4] "Sistema ***single-user***" neste contexto é usado para descrever cenários de uso de sistema, não capacidades. O sistema pode ter capacidade de "*multi-user*", mas se o cenário testado é usado apenas por um único usuário, o comportamento do escalonador será avaliado como um sistema *single-user*.

alcançados via uma taxa de linha de tendência de 1,25, para baixo valores da função *nice()*.

Este primeiro escalonador tem como objetivo primário a construção de um *framework* de testes para verificar e comparar o desempenho de escalonadores. Seus poucos resultados estão em conformidade com os aspectos teóricos de escalonador ***CFS***, cuja expectativa é encontrar uma relação linear, por construção, entre o tempo total de execução de uma tarefa e o valor da função *nice()*. Esta relação é de $1{,}25^{-(nice)}$, (como visto na Seção 3.1.1 desta dissertação) e é corroborada pelos dados experimentais e conclusão dos autores. Apesar da excelente elaboração da formulação teórica para abordagens de testes de escalonadores no caso geral, o artigo não visa discorrer especificamente sobre os detalhes de implementação do ***CFS*** e sim limitar-se a usar o escalonador ***CFS*** padrão com os parâmetros *default* do escalonador como mero exemplo da aplicação do *framework* de testes proposto. Diferentemente da abordagem do trabalho fruto desta dissertação, que leva em conta vários experimentos com dois escalonadores, ***O(1)*** e ***CFS***, e também parametriza as principais variáveis de escalonador como *sched_latency_ns, sched_min_granularity_ns e sched_wakeup_min_granularity_ns*. O objetivo do primeiro trabalho correlato é propor, elaborar e exibir um conjunto de testes para verificação e comparação de escalonadores em geral. Estes testes são aplicados a um estudo de caso muito simplificado do ***CFS*** em particular com resultados modestos, que não podem ser extrapolados por terem sido elaborados em pequena escala apenas. Pode-se concluir que o primeiro trabalho é mais uma prova de conceito do que um avaliação formal.

O segundo trabalho, "*BFS vs. CFS - Scheduler Comparison*" [74]**,** compara diretamente o desempenho de dois escalonadores para o Linux: o ***CFS***, integrado na versão do kernel Linux 2.6.23 e a nova proposta de Con Kolivas com o escalonador ***BFS*** (*Brain Fuck Scheduler*), em sua versão 3.3.1. de 2009. Inicia identificando o que considera os principais aspectos de comparação entre os escalonadores, isto é: a importância de maximização de CPU; o quão saturada é a utilização de CPU de 0 a 100%; a maximização de *throughput* – número de processos completados por unidade de tempo; a minimização do *turnaround* - o tempo total que um processo leva até ser completado, do ínicio ao fim; o tempo de espera – o tempo total que um processo gasta na fila de prontos; a redução da latência do escalonador – o tempo decorrido entre um sinal enviado para acordar uma *thread* na fila de espera e aquele que o escalonador levou para liberar a execução daquela *thread*. Executar o escalonador mais

frequentemente reduzirá a latência do escalonador assim a *thread* gastará menos tempo esperando no estado de pronto. Entretanto existe um dilema pois executar mais frequentemente o escalonador implica em gastar mais tempo de gerenciamento e menos tempo na execução da tarefa propriamente. Idealmente, o escalonador executará tão rápido quanto possível, após ter ocorrido uma interrupção que acorde a *thread*.

Um bom algoritmo de escalonamento depende do tipo de trabalho que está sendo feito na máquina. Em geral, este trabalho pode ser classificado em: *batch*, *interativo* e *tempo real*.

Muito dos escalonadores comercializados são filas de retroalimentação de múltiplos níveis. Isto inclui os escalonadores do Linux anteriores a versão 2.6.23, bem como as versões Windows baseadas no NT tais como Windows XP, Vista e 7. Estes escalonadores usam filas separadas para dar preferência a processos interativos e I/O intensivos. Com filas de retroalimentação de múltiplos níveis, novos processos são inicialmente enfileirados em uma fila de média prioridade. Se a tarefa usa sua fatia de tempo inteira, então ele é colocado numa fila de prioridade mais alta. Dependendo da implementação, para compensar processos I/O intensivos, os processos em filas de prioridade mais altas podem ter mais fatias de tempo e serem escalonados antes dos processos das filas de mais baixa prioridade.

Observa-se, de maneira significativa, que o escalonador ***BFS*** possuí uma única fila de controle para todos os processos e processadores e um conceito de um *deadline virtual* (*vdeadline)* relativo a cada CPU, ou seja, não existe ordenação absoluta dos processos através de seus *deadlines* virtuais. Portanto; os processos não podem ser colocados numa árvore.

No que se refere aos experimentos, observa-se que os dois escalonadores foram compilados independentemente em duas cópias com o kernel na versão 2.6.31.6. O teste de execução foi realizado com as ferramentas: **latt.c** para aferição dos tempos de latência e *turnaround* do escalonador sob várias cargas do sistema; e **make** para focar-se nos testes do tempo de *turnaround* dos processos; e, e testes para mostrar a latência na exibição de vídeos.

Os resultados experimentais sobre os tempos de resposta, apresentados sobre a latência de clientes *notebooks*, tempos de *turnaround* de clientes de máquinas *desktop* multicore, compilação via **make** em *desktops* e da interatividade de exibição de vídeos via Mplayer, não garantem as conclusões devido ao fato de terem sido realizados em pequena escala apenas (poucos *cores* e clientes); portanto, não é possível extrapolar as

conclusões para a grande escala (muitos processadores e clientes) – como visto nesta dissertação. Dos resultados obtidos a partir dos quatro experimentos, verifica-se, segundo os autores, que o ***CFS*** supera o ***BFS*** na minimização de *turnaraound* e que é superado pelo ***BFS*** na minimização de latência. Isto indica, segundo os autores, que o ***BFS*** seria melhor para tarefas interativas que bloqueem I/O ou entrada de usuário e que o ***CFS*** é melhor para processamento em batch, que é CPU intensivo.

Quanto aos resultados do segundo trabalho, quando comparados com os dados desta dissertação, revelam nitidamente ser o ***CFS*** mais interessante, devido ao seu caráter geral, diferentemente do ***BFS***, que é especifico e localizado em desempenho de *desktop* e *workstations* ou PCs. As principais características e funcionalidades do ***CFS*** e ***BFS*** são reveladas grosso modo. Todos os três trabalhos não atuam sobre as variáveis dinamicamente reconfiguráveis do escalonador ***CFS***. Portanto, a única conclusão favorável do trabalho sobre a superioridade do ***BFS*** sobre o ***CFS***, conforme os resultados dos dados testados, poderia ser confrontada para o ***CFS*** configurado para tarefas interativas, ou seja, configurando-se o parâmetro *sched_min_granularity_ns* para baixos valores.

O terceiro trabalho, "*A comparison of interactivity in the Linux scheduler and an MLFQ scheduler*" [75]**,** compara a interatividade de dois escalonadores de sistemas distintos: o escalonador do Linux kernel 2.6, ***O(1)***, e outro do tipo **MLFQ (***Multi Level Feedback Queue***)** incorporado ao kernel do Linux. Apresenta as principais características do escalonador do Linux 2.6, ***O(1)***, e de seus algoritmos com suas duas filas "ativa" e "expirada" por CPU.

O escalonador **MLFQ** é descrito suscintamente como tendo múltiplas filas, usualmente filas *round-robin* com prioridades e fatias de tempo associadas a cada uma delas. O tamanho da fatia de tempo é inversamente proporcional ao nível da prioridade. Um processo é movido para uma fila de prioridade mais baixa se usar toda a sua fatia de tempo, e executa por um intervalo de tempo mais longo da próxima vez que detiver o controle da CPU; é movido para uma fila de prioridade mais alta se for bloqueado antes de completar toda a sua fatia de tempo, e executa por um intervalo de tempo menor da próxima vez. O escalonador escolhe um processo da fila de prioridade mais alta não vazia para execução, e um processo que chega com prioridade mais alta causa preempção num processo de prioridade mais baixa em execução. A seguir, o trabalho discorre sobre o mecanismo de tomada de decisão do **MLFQ** e compara diretamente os algoritmos dos escalonadores em estudo, isto é, ***O(1)*** e **MLFQ**. São descritos em

detalhes os testes de interatividade do **MLFQ** e aplicados numa máquina uniprocessada com cargas de trabalho CPU-intensivas e de compilação. Dados provenientes dos testes anteriores são exibidos para comparação revelando valores muito próximos para os testes de CPU-intensiva e com significativos ganhos para o **MLFQ** no caso cargas de trabalho de compilação, que são de natureza CPU intensiva e paralelizáveis na maioria das vezes pois podem ser compilados módulos ou programas distintos concorrentemente.

Não podem ser comparar os resultados do terceiro trabalho com os obtidos nesta dissertação pois não são apresentadas as variâncias das medidas no terceiro trabalho embora os dois tenham testado o escalonador ***O(1)*** em comum.

O quarto trabalho**,** "*Application of Global and One-Dimensional Local Optimization to Operating System Scheduler Tuning*" [76]**,** faz um estudo comparativo do ajuste fino do escalonador ***CFS*** do Linux aplicando os métodos global e local de otimização a uma dimensão. Basicamente, são executadas simulações de uma aplicação ou *workload* sob o ***CFS*** variando-se os seguintes parâmetros do escalonador: *sched.latency_ns*, *sched.min_granularity_ns* e *sched.wakeup_min_granularity* no intervalo de 100.000 a 1.000.000.000 ns os dois primeiro e de 0 a 1.000.000.000 ns o terceiro.

O trabalho conclui que é possível otimizar os parâmetros do escalonador CFS do Linux a fim de obter valores ideais para a minimização do *turnaround* de uma aplicação ou do número mínimo de iterações necessárias de maneira sistemática.

O quarto e último trabalho, corrobora esta dissertação, onde também é mostrado de maneira empírica, a viabilidade de otimização do escalonador ***CFS*** para aplicações ou *workloads* específicos através da reparametrização dinâmica das principais variáveis do ***CFS***: *sched.latency_ns, sched.min_granularity_ns e sched.wakeup_granularity_ns*. Particularmente, nesta dissertação, os dados de avaliação de testes comparativos entre os escalonadores ***O(1)*** e ***CFS*** foram gerados com medições de tempos de processamento com aplicações **LI** e **CF,** diferentemente da abordagem do quarto trabalho contrariamente aos dados do quarto trabalho que medem o tempo mínimo de *turnaround* de mecanismos de otimização local e global, e o número de iterações mínimo para se obter este *turnaround*.

Cabe ressaltar que o tratamento de *CGroups* é exclusivo desta tese e não faz parte do escopo dos outros trabalhos correlatos apresentados.

2.8 RESUMO

Neste capítulo, a partir do conceito de escalonadores de processos em geral foram descritos cronológica, funcional e estruturalmente os principais escalonadores do Linux desde sua origem em 1991 até os dias atuais. Foram apresentadas as características do escalonador ***CFS*** e o significado do novo conceito do *virtual run time* (*vruntime*) e como ele é calculado no ***CFS***. A anatomia do escalonador ***CFS*** do Linux e suas principais estruturas e métodos são descritos no ANEXO A. Outros trabalhos correlatos são brevemente citados para fins de comparação.

CAPÍTULO 3 – DETALHAMENTO DO CSF

Como visto no capítulo anterior, o escalonador ***CFS*** (*Completely Fair Scheduler*) foi inserido no *kernel* 2.6.23 do Linux com algumas melhorias. Logo a seguir, no *kernel* 2.6.24, incluída a biblioteca que implementa o *CGroups* (*libcgroups*) para controle e particionamento de recursos compartilhados por usuários, grupos de usuários ou aplicações. Este capítulo, apresenta uma explicação mais detalhada da heurística ***CFS*** e os primeiros experimentos realizados para avaliar o ***CFS*** em comparação com o escalonador anterior, ***O(1)***, isto é, a comparação do escalonador do Linux **com** e **sem** o ***CFS***.

3.1 DETALHAMENTO E SÍNTESE DO *CSF*

A proposta fundamental do escalonador ***CFS*** é de implementar uma divisão equânime ou justa do tempo de processador para os diversos processos que compartilham o sistema, diferentemente do escalonador anterior ***O(1)***, como pode ser visto em [30] [77]. A equidade é um dos objetivos dos escalonadores de processos. Através dela, impede-se que processos fiquem em estado de inanição (*starvation*) em uma demorada espera por recursos do sistema [34] como, por exemplo, CPU.

O *kernel* do Linux com o seu atual escalonador ***CFS*** permite total compatibilidade com as arquiteturas Intel/AMD x86_32 e x86_64, IBM PowerPC adequado para os diversos tipos de equipamentos como *netbooks*, *notebooks*, *workstations* e servidores constituídos de processadores com estas arquiteturas além de outros embarcados e diversos dispositivos móveis. A compatibilidade do *kernel* é garantida porque o mesmo foi desenvolvido em linguagem de alto nível, linguagem C, e os fontes podem ser recompilados para qualquer uma delas [78]. Desta forma, um único *kernel*, com escalonador adequadamente parametrizado, pode servir às diversas plataformas, evitando assim a implementação de um *kernel* para cada sistema de pequeno, médio ou grande porte como no caso do Windows (WindowsCE, Windows Server, Windows 7, Windows Home Edition, etc).

Além disso, como é nosso objetivo verificar, ao ser mais justo em relação à divisão de tempo de CPU, o tempo de resposta entre as aplicações tende a diminuir, favorecendo assim uma maior interatividade entre as mesmas e melhorando a percepção de interação entre os usuários do sistema como um todo, através de suas sessões de

login ou terminal, por exemplo, e as trocas entre as diversas aplicações *online*, através da movimentação do mouse, por exemplo.

O ***CFS*** foi criado pelo desenvolvedor Ingo Molnar e evoluiu inspirado no escalonador anterior do Linux denominado ***O(1)*** e no ***RSDL***. O pré-requisito para esta evolução consiste na fundamentação das rotinas de temporizadores (*timer)* de alta resolução (*hrtimer*), implementados por Thomas Gleixner [69] [70] que permitem tratar com tempos com precisão de nanossegundos, graças a implementação por hardware da tecnologia APIC (Advanced Programmable Interrupt Controller) da Intel [79]. Outro pilar fundamental foi o abandono a qualquer *HZ* ou *jitter*, graças ao sistema operacional *tickless* (como pode ser visto no Apêndice A), tornando o escalonador livre de interferências constantes e pré-definidas limitações de hardware, além de não implementar complexas heurísticas para as decisões de escalonamento das tarefas. Praticamente, os tempos de CPU são repartidos aritmeticamente entre os *n* processos que dele dependem, de tal forma que cada um deles, conforme proposta de um escalonador ideal, obtenha 1 (*um) n-ésimo* do tempo total disponível. Por isso, conceitua-se de heurística simples.

Outra alteração fundamental em relação aos anteriores é mudança nas estruturas de controle do escalonador. O escalonador ***CFS*** não se estrutura em listas encadeadas para cada prioridade do sistema como seus antecessores, mas sim em uma árvore binária rubro-negra para cada processador ou *core* físico ou lógico (quando o processador possui tecnologia HT *(Hyper-threading)* ou SMT *(Simultaneous multi-trheading)* habilitadas) ou *cgroup* definido. A escolha de uma estrutura em árvore binária rubro-negra facilita a atualização (exclusão e/ou inclusão de um nó na árvore), pois as árvores binárias rubro-negras são consideradas autobalanceadas[5] [38] embora com custo de $O(log\ n)$, onde n é o número de nós da árvore, que representa o número total de processos do sistema naquela fila de execução correspondente, ou seja, naquela CPU. Dado que n é geralmente pequeno, o custo de atualização das árvores binárias rubro-negras, que representam as filas de execução de cada CPU, torna-se desprezível para estes valores de n (da ordem de poucas dezenas de processos por fila de execução).

Em cada nó, de cada árvore, de cada CPU física ou lógica temos o PID do processo e um valor associado a cada nó denominado *vruntime* do processo, que

[5] Uma árvore de busca binária é dita autobalanceada quando as alturas das duas sub-árvores a partir de cada nó diferem no máximo em uma unidade. As operações de busca, inserção e remoção de elementos possuem complexidade $O(\log_2 n)$ (no qual n é o número de elementos da árvore). Inserções e remoções podem também requerer o rebalanceamento da árvore, exigindo uma ou mais rotações.

identifica um tempo que cada processo individualizado consumiu desde o início de sua execução no sistema. Este tempo não é real. Ele é um "tempo virtual" de um processo específico de determinado PID, pois representa o tempo que o processo consumiria em uma máquina ideal, considerando que um processo, por exemplo, pode ser migrado para qualquer CPU existente ou aguardar na fila de execução por CPU. Quando a árvore binária rubro-negra é atualizada, o processo do nó mais à esquerda da árvore rubro-negra é aquele de menor *vruntime* (resultado da ordenação da árvore rubro-negra, da árvore binária balanceada pela chave *vruntime* que representa a fila de processos para cada CPU) e este, é o candidato a ser o próximo a executar naquele processador. Esta é forma pela qual o escalonador ***CFS*** sempre tomará sua decisão de qual processo será o seguinte a executar naquela fila de execução. Já o nó mais à direita da árvore rubro-negra representa o processo com maior *virtual run time*, isto é, o processo com maior tempo de execução daquela fila de execução, em outras palavras, aquele que irá aguardar na fila até que todos os outros com menor *vruntime* daquela fila tenham tido acesso a CPU.

O ***CFS*** também repetiu o modo de partilha de tempo do sistema anterior, ***O(1)***, dividindo em uma época ou período de tempo máximo, associada às latências do escalonador, subdividida em fatias menores imaginadas do tamanho de tempo a ser dado a cada tarefa para um número de tarefas igual a 5 por padrão. Estes conceitos estão associados às variáveis dinamicamente reconfiguráveis do escalonador denominadas *kernel.sched_latency_ns* e *kernel.sched_min_granularity_ns*, dadas em nanossegundos (ns) que podem ser modificados com o sistema no ar pelo administrador (root) ou na perda de recurso como, por exemplo, desativação de uma ou mais CPUs. Os conceitos anteriores de época ou período, ou *time slice* fixos desaparecem.

3.1.1 Detalhamento do funcionamento do algoritmo do CFS

Algoritmo do CFS do Linux e Runtime Virtual.

Como citado anteriormente, o ***CFS*** utiliza uma árvore rubro-negra como estrutura de dados e ao contrário dos escalonadores de processos anteriores do Unix que levam em conta uma fatia de tempo tradicional de execução do processo, considera o que é referido como o tempo virtual de execução do processo (*virtual runtime*), expresso em nanossegundos (ao contrário de HZ ou *jiffies*). O uso de uma árvore autobalanceada como a árvore rubro-negra permite uma pesquisa de complexidade de

tempo da ordem de *O*(log *n*) por altura da árvore, onde *n* é o número de nós da arvore rubro-negra, que representa a quantidade de processos na fila de execução de uma dada CPU [87].

Virtual Runtime

O tempo virtual de execução (*virtual runtime* [30] [44] [43]), declarado na variável "*vruntime*" é uma parte variável do processo herdado da estrutura C, tal como definido em *linux / sched.h*, que representa o tempo que um processo executou em relação ao número de processos na fila de execução. Este tempo de execução virtual é o tempo ponderado da execução de um processo na CPU cujo peso é dado pelo peso do processo com valor *nice* 0 (zero) dividido pelo peso do processo corrente, ou seja, o peso depende do valor *nice* do processo e é especificado, aproximadamente, por $1024*(1{,}25)^{-(nice)}$.

Enquanto as outras variáveis também desempenham um papel importante no algoritmo de tomada de decisão do ***CFS***, *vruntime* é, de longe, a variável do *kernel* que precisa de mais atenção para se compreender o processo de decisão do escalonador. Como afirmado anteriormente, o ***CFS*** não leva em conta a fatia de tempo do processo, como nos escalonadores antigos. Ao invés disto, o *vruntime* é avaliado. O *vruntime* para cada processo é calculada como segue:

1 – Calcula-se o tempo gasto pelo processo na CPU
2 - Calcula-se o tempo ponderado de execução em relação ao número de processos na fila de execução para cada CPU ativa

A função *update_curr* é responsável por calcular o tempo de execução gasto pelo processo, que armazena o valor em uma variável unsigned long "delta_exec". A variável "delta_exec" é calculada como segue:

```
delta_exec = (unsigned long) (now - curr->exec_start)
```

"Delta_exec" é passado para outra função, *update_curr()*, que no seu interior, chama a função *calc_delta_fair()*, responsável pela determinação do tempo de execução ponderado do processo e depende do tempo de execução ponderado pelo valor da

prioridade do processo dado pela função *nice*(). Lembramos que no ***CFS*** não existe o conceito de prioridade para os processos da classe *fair* e que os valores da função *nice*() são somente para efeito de uma compatibilização com escalonador anterior com uma certa linearidade diretamente proporcional aos valores desta função *nice*(), e dado por:

```
delta_exec_weighted = calc_delta_fair (delta_exec, Curr);
```

Logo após, vruntime do processo é atualizado:

```
curr->vruntime + = delta_exec_weighted;
```

Finalmente, é executada uma função que calcula o tempo virtual mínimo dentre todos os processos peertencents a mesma fila de execução, pois ele será colocado no nó mais à esquerda da árvore rubro-negra e, portanto será o próximo processo a ser escalonado, isto é, passar a deter o controle de processamento daquela CPU correspondente.

```
update_min_vruntime (cfs_rq);
```

"*Calc_delta_fair()*" irá retornar o valor ponderado de tempo de execução calculado pelo processo, "delta_exec", em relação ao número de processos executáveis. A função utilizada para esse cálculo é "*calc_delta_mine*".

```
calc_delta_mine()
```

Uma vez que o *vruntime* é calculado, é armazenado na estrutura *sched_entity* herdada do processo, chamando

```
curr->vruntime + = delta_exec_weighted;
```

na função anteriormente vista *update_curr()*, também notar a chamada

```
update_min_vruntime (cfs_rq);
```

O que esta função faz é calcular o menor *vruntime* de todos os processos executáveis e armazená-lo no nó mais a esquerda (*rb_leftmode*) da árvore rubro-negra, onde residem a identificação do processo de menor *vruntime* e o valor de seu *vruntime*.O algoritmo de seleção de ***CFS*** para o processo a ser executado é extremamente simples, ele irá executar a sua árvore rubro-negra e pesquisar o menor valor *vruntime* apontando para o próximo processo executável. Em outras palavras, "executar o processo que é representado pelo nó mais à esquerda na árvore rubro-negra", uma vez que o nó mais à esquerda contém o menor *vruntime*, referido no código-fonte como *min_vruntime*. Para concluir, é importante notar que a ***CFS*** não percorre toda a árvore desde que *min_vruntime* é referenciado por *rb_leftmost* no **cfs_r*q*** *struct* (kernel / sched.c), isto é, no conteúdo do nó mais à esquerda da árvore rubro-negra que representa a fila de execução de uma CPU.

3.2 METODOLOGIA DOS EXPERIMENTOS COM OU SEM O CFS

Duas estratégias ou abordagens práticas serão usadas nos experimentos desta seção. A primeira visa estudar qualitativamente o comportamento do tempo de resposta do escalonador para executar processos concorrentes em relação ao aumento da carga devido ao crescimento da quantidade de processos a serem executados concorrentemente. Neste caso, é verificado se o tempo compartilhado de cada processo é proporcionalmente equitativo em relação ao número de processos concorrentes naquele instante de amostragem. Espera-se que os gráficos representem e exibam suaves tendências e regularidades em relação à distribuição de tempos de CPU pelos processos naquele intervalo de amostragem.

A segunda estratégia, objetiva estudar o mesmo comportamento de ponto de vista quantitativo, ou seja, dados vários processos com uma determinada e definida carga de trabalho, é medido o tempo de CPU de cada processo em um dado intervalo de tempo, para que se verifique em quanto tempo real ou de parede (*wall-clock)* os processos são efetivamente executados. Assim, acredita-se poder avaliar a equidade proposta pelo escalonador ***CFS*** em relação ao escalonador anterior, ***O(1)***, pela análise

da variância[6] dos tempos de CPU de cada processo em relação aos seus valores médios em cada intervalo de tempo para cada escalonador: **com** e **sem** o ***CFS***.

3.2.1 Com *loop infinito* (queima indefinida de segundos de tempo de CPU)

Para a primeira estratégia, de *loop infinito*, doravante denominada **LI**, escrevemos um simples programa em linguagem C que executa em um *loop infinito*: a soma de uma constante em uma variável inteira no interior do *loop* (não há preocupações com *overflow*, pois a variável da soma é definida como inteiro sem sinal, logo, em uma soma infinita, retorna ao seu valor inicial zero e prossegue a adição sem maiores problemas). O simples código fonte pode ser visto abaixo:

```
main()
{
  unsigned long int i = 0;
  while(1) {
    i++;
  }
}
```

Ao processo associado ao pequeno módulo de processamento acima, denominaremos P_1 e dispararemos outros processos P_2, P_3, ..., P_n a cada intervalo de tempo subseqüente em segundos, ou seja, P_1 no tempo t=0, P_2 em t=1, P_3 em t=2, e assim por diante até submetermos o *n-ésimo* processo P_n.

Todos os PIDs (identificadores de processo – "*Process Identification*") dos processos P_i´s disparados acima são identificados com o comando *pidof* do Linux dentro do script em linguagem *bash* (interpretador de comandos) e para cada um destes PIDs são verificados os tempos de execução via o comando /proc/$pid/stat em um arquivo próprio para cada PID até um tempo limite de execução para todos os jobs submetidos em *batch* (*backgroud*) no *script* que os aciona (comando terminados por &). Todos os arquivos resultantes da execução dos jobs são guardados para tratamento adequado em tabelas ou gráficos, visando a revelação qualitativa e/ou quantitativa da proposta do escalonador ***CFS***. Verifica-se se o escalonador é qualitativamente e quantitativamente justo ou não, conforme sua proposta fundamental. Isto será confirmado se o compartilhamento do tempo de CPU for proporcional a 1/*n*, onde *n* é o

[6] A **variância** de uma variável aleatória é uma medida da sua dispersão estatística indicando quão longe em geral os seus valores se encontram do valor esperado.

número de processos, ou seja, se as porções efetivas de tempo de CPU de cada processo são aproximadamente iguais a cada intervalo de tempo. Espera-se que, graficamente, as hipóteses possam ser reveladas pelo comportamento suave da distribuição das cores, que representam os tempos de cada processo em um intervalo definido, variando-se a quantidade de processos por unidade de tempo e a aparente proporcionalidade equitativa dos segmentos coloridos esperada.

3.2.2 Com carga definida (queima de 500.000.000 adições de uma constante em uma variável inteira)

A segunda estratégia consiste em determinar e definir uma carga de trabalho $\mathbf{W}_{CPU}$ fixa como sendo a carga correspondente a um *loop* finito correspondente a 500 milhões de adições de uma constante em uma variável inteira ou, basicamente, queima do tempo de CPU necessário para servir como padrão de execuções concorrentes de processos em paralelo e com muitos destes, coletando-se os tempos efetivos ou de parede (*wall-clock)* de cada um dos processos. Em outras palavras, o tempo efetivo que cada processo leva para executar a tarefa fixada. O objetivo deste método, doravante denominado carga fixa, **CF**, é o mesmo do anterior, isto é, possibilitar a avaliação e verificação se o escalonador ***CFS*** é realmente justo em relação à divisão de tempo de CPUs entre os processos concorrentes em execução em um mesmo *host* à medida que aumentamos o número de processos concorrentes no sistema.

O tempo de execução é de, aproximadamente, 1 segundo, mas este valor não é importante, pois o que é mensurado é o tempo da carga fixa de CPU necessário para executar um loop com uma carga fixada de 500 milhões de operações idênticas. O simples código fonte pode ser visto abaixo:

```
main()
{
/* maxCF é o numero máximo de cargas por processo ~1
sec */
  unsigned int i, j, MaxCF=1;
  for ( i = 1; i <= maxCF; i++ ) {
    for ( j = 1; j <= 500000000; j++ ) { }
  }
}
```

Para os experimentos, são usadas plataformas ou computadores baseados em processadores Intel *multicore* (um *notebook* dual *core* comum, uma *mobile workstation*

dual *core*, outra *mobile workstation quadcore* e uma *workstation* dual *processor hexacore*). Para os sistemas operacionais destas plataformas de hardware, temos o CentOS *kernel* Linux 2.6.18 ou 2.6.19 e o Red Hat Enterprise Linux 6, ou seja, *kernel* Linux 2.6.32 instalados em todas as máquinas com *dual boot*. Para fins práticos, será denotado o sistema operacional com escalonador de *kernel* **com** ou **sem *CFS***, respectivamente aos sistemas operacionais anteriores.

A razão da utilização dos dois sistemas acima é simples: como visto anteriormente, o escalonador ***CFS*** (*Completely Fair Schedule*) foi inserido no Linux a partir do *kernel* 2.6.23 e com melhorias no 2.6.24 com a inclusão dos grupos de controle (*Control Groups*). Portanto, trabalharemos com *kernels* do Linux **com** e **sem** o ***CFS*** para efeito de comparação da distribuição de tempos de CPU entre os processos concorrentes em cada sistema operacional para as várias plataformas acima descritas. Logicamente, para os experimentos com os grupos de controle (*Control Groups*), será usado apenas o *kernel* **com *CFS***.

Para a obtenção das estatísticas de tempo de processamento tanto de **CPU** efetiva como tempo de parede (*wall-clock),* são usadas duas abordagens: uma através das estatísticas geradas pelo próprio sistema operacional no pseudo *filesystem* /proc [80], no arquivo /proc/stat ou no /proc/$pid/stat; e outra através das diferenças de tempo real obtidas pelo relógio real do sistema via rotinas do *kernel* como *clock_gettime*(REALCLOCK, null), que nos permitiu uma maior precisão e confiabilidade dentro dos próprios processos, sendo exportados para o mundo externo via arquivos do tipo texto identificados univocamente (na abordagem inicial dos trabalhos).

3.3 RESULTADOS PRELIMINARES

Nesta seção, serão apresentados dois experimentos cada um com aplicação do método **LI** ou **CF, com** ou **sem** a presença do escalonador ***CFS***. Inicialmente, os experimentos irão lidar com objetos fundamentais tratados pelos escalonadores (CPU e processos) em pequena escala: duas CPUs e poucos processos (até dezena). O objetivo primário deste experimento é buscar e identificar um comportamento singular que permita inferir hipótese sobre os escalonadores em estudo, sobretudo o CFS. São realizadas análises correspondentes a cada experimento e conclusões sobre as diferenças entre eles.

3.3.1 Avaliação do escalonador em uma pequena escala

A seguir, usando a estratégia **CF** aplicada a pequena escala (poucos processos, CPUs e tempo de execução)**,** exibiremos os resultados gráficos da submissão concorrente de 2, 3, 4, 5, 7 e 10 processos com cargas fixas de 1 segundo e 10 segundos, (modificando-se o MaxCF), respectivamente nos sistema **sem** e **com** o ***CFS***. Os experimentos são apresentados em grupos de três para efeito de simplificação e clareza, pela ordem, ou seja, inicialmente os de 1 segundo de **CF** (carga fixa) **sem** o escalonador ***CFS*** e seus respectivos grupos correspondentes ao sistema **com** o ***CFS***. Todos os dados foram obtidos com *notebook dual core*.

Grupo com 2, 3 ou 4 processos concorrentes de 1 segundo de **CF sem *CFS***

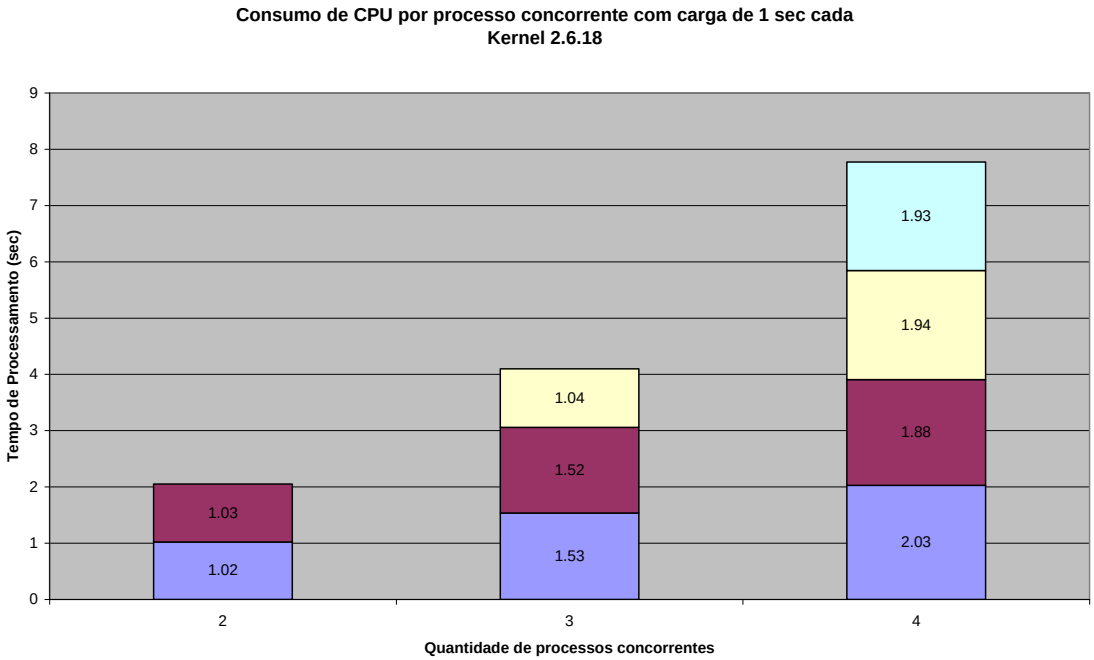

Figura 3-1 Gráfico do consumo de CPU por processo concorrente. Sem *CFS*.

Grupo com 2, 3 ou 4 processos concorrentes de 1 segundo de **CF com *CFS***

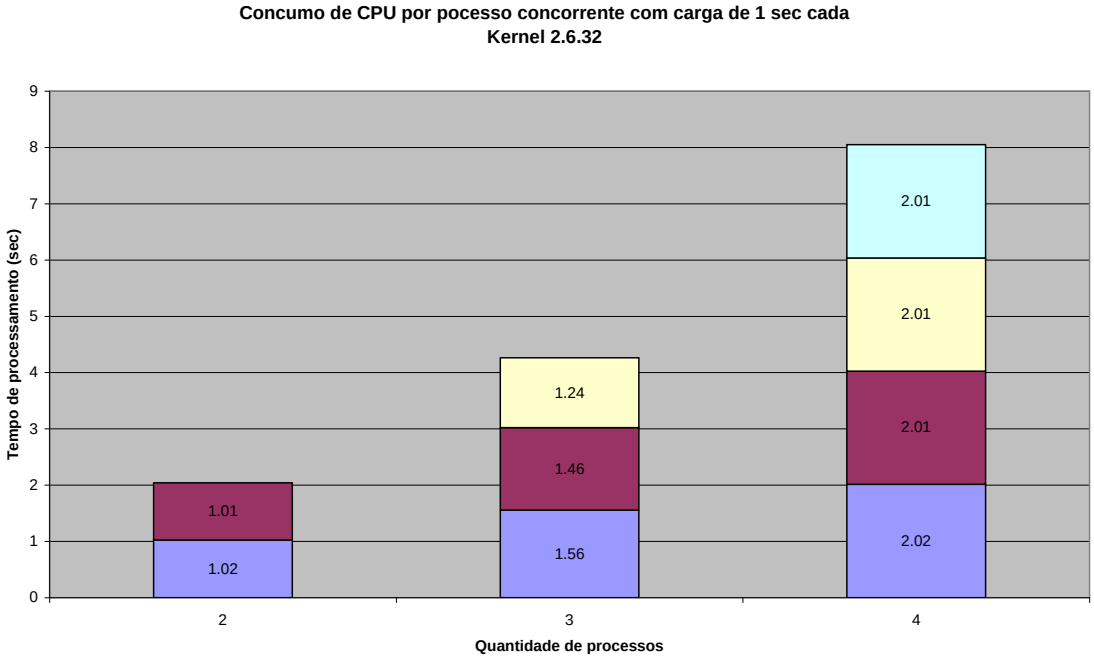

Figura 3-2 Gráfico do consumo de CPU por processo concorrente. Com *CFS*.

Os gráficos da Figuras 3-1 (**sem *CFS***) e da Figura 3-2 (**com *CFS***) revelam que, mesmo para pouquíssimos processos, observam-se significativas diferenças na distribuição de tempo de CPU entre os processos. Quando o número de processos é idêntico ao de CPUs, nada a ser observado pois, existem recursos disponíveis para todos os processos e os escalonadores **sem** e **com *CFS*** se comportam de modo idêntico. Entretanto, quando o número de processos aumenta, por menor que seja este aumento, o escalonador **sem *CFS*** tende a completar os processos em menor tempo do que o escalonador **com *CFS***. O mesmo se observa para uma quantidade de processos múltipla do número de CPUs. É compreensível à luz do funcionamento dos escalonadores. O escalonador **sem *CFS*** entrega *slices* de tempo muito maiores que as granularidades mínimas do ***CFS***. Implica que o ***CFS*** necessita efetuar muitas entregas de pequenas fatias de CPU para todos os processos concorrentes mais vezes, aumentando assim o tempo total para o processamento de todos os processos. Por outro lado, tende a aproximar os tempos de término de cada processo já que os recursos são distribuídos de modo o mais igualitário possível. Os valores de CPU do escalonador **sem *CFS*** são mais baixos do que os **com *CFS***, ou seja, de melhor desempenho mas, nitidamente não justos pois os valores de cada processo são bastante diferentes. Em uma conclusão precipitada pode-se optar pelo ***sem CFS*** para efeito de desempenho. Mas, são necessários mais testes com maiores tempos de CPU, como veremos mais a frente.

Para as variâncias, temos os seguintes valores para cada escalonador:

CFS	Número de Processadores		
	2	3	4
sem	0,000045	0,079288	0,004010
com	0,000044	0,026168	0,000007

Tabela 3-1 Variâncias em pequena escala com CF = 1 segundo

Os valores das variâncias da primeira parte do experimento só confirmam as hipótese anteriores pois, o escalonador **com *CFS*** possui os mais altos valores de CPU

dos processos porém mais próximos uns dos outros nos três casos. Aqui, o **com *CFS*** leva uma desvantagem por sua busca por justiça a custa de um número maior de trocas de contexto, possivelmente como se pretende mostrar.

Nitidamente, verifica-se que a variância para os valores com ***CFS*** são muito menores. Portanto, mais uma vez, podemos concluir, até aqui, que o escalonador ***CFS*** tende a ser menos injusto que o seu anterior para uma máquina com poucas CPUs. Mas, é precipitado concluir e generalizar pois existem pouquíssimas amostras. Mas, tanto do ponto de vista qualitativo (análise das regularidades ou irregularidades das distribuições proporcionais das cores dos gráficos, cujos segmentos representam os tempos de cada processo no eixo das ordenadas em função do número de processos concorrentes no eixo das abscissas) quanto do quantitativo (pelo estudo das variâncias para cada conjunto de mesmo número de processos concorrentes) há boas indicações que o ***CFS*** pode vir a ser realmente menos injusto que seu antecessor.

Grupo com 5, 7 ou 10 processos concorrentes de 1 segundo de **CF sem *CFS***

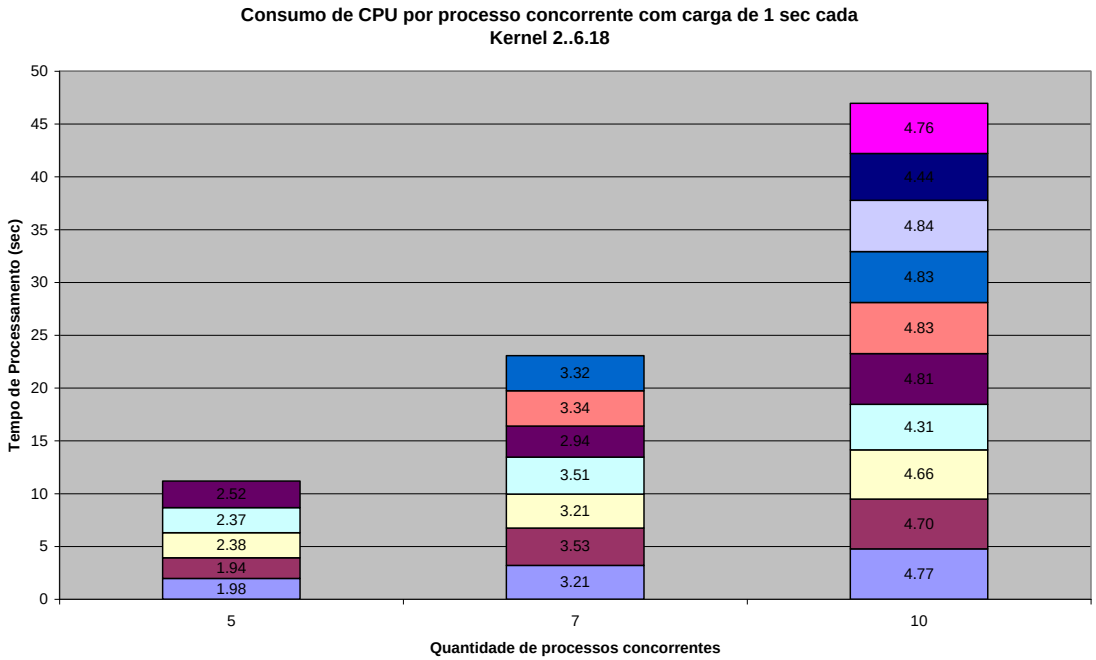

Figura 3-3 Gráfico do consumo de CPU por processo concorrente. Sem *CFS*.

Grupo com 5, 7 ou 10 processos concorrentes de 1 segundo de **CF com *CFS***

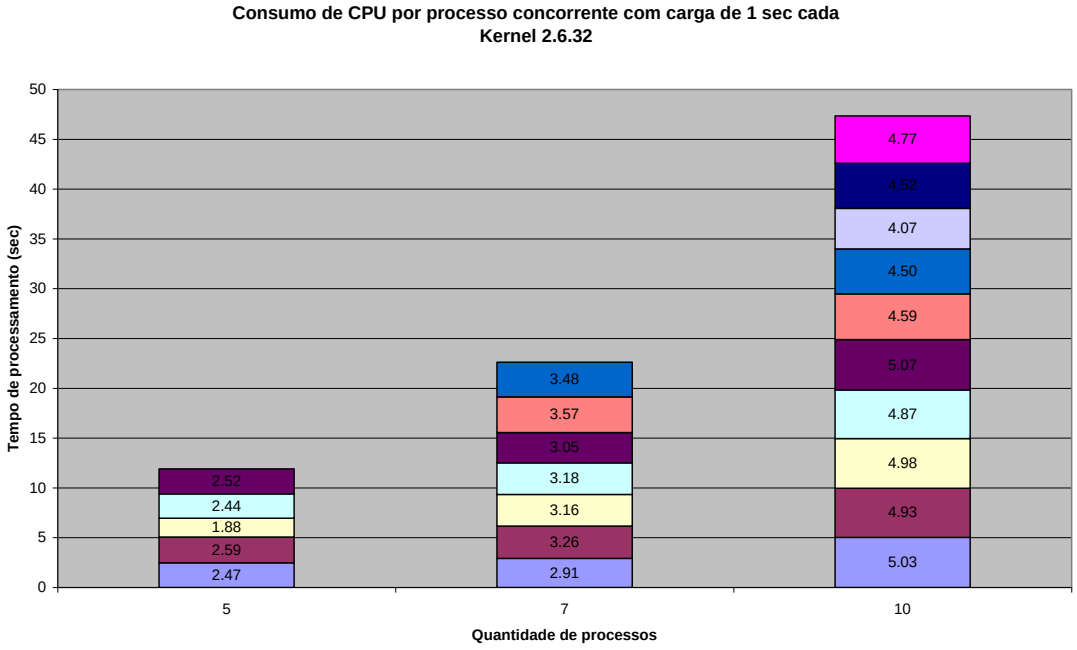

Figura 3-4 Gráfico do consumo de CPU por processo concorrente. Com *CFS*.

A análise qualitativa dos gráficos correspondentes às Figuras 3-3 e 3-4, respectivamente com 5, 7 e 10 processos de 1 segundo, nos revela que a disparidade na distribuição de CPU entre os processos tende a aumentar com o número de processos para o escalonador anterior ao ***CFS.*** Pelas considerações anteriores, isto parece óbvio pois com o aumento do número de processos e diretriz por justiça do ***CFS***, implicará em um custo mais alto ainda pois mais processos receberão menores fatias de processamento um número de vezes maior ainda.

CFS	Número de Processadores		
	5	7	10
sem	0,067864	0,039743	0,032709
com	0,080947	0,053485	0,097389

Tabela 3-2 Variâncias em pequena escala com CF = 1 segundo

Observando os dados, deveríamos ser levados a concluir que, neste caso, o escalonador anterior seria menos injusto, mas devemos notar que criador do ***CFS*** afirma que ele é 80% justo. Então, até aqui, nada se pode afirmar categoricamente.

Por outro lado, como será mostrado mais a frente neste trabalho, o escalonador entrega mais frações de tempos menores de CPU para os processos de maneira mais frequente. Isto é, tende a gastar mais tempo com as trocas de contexto necessárias para estas distribuições em maior número. Em outras palavras, sob certas circunstâncias muito específicas, o escalonador anterior pode vir a ser aparentemente mais justo. Entretanto, devemos perceber que apenas o ***CFS*** é dinamicamente reconfigurável. Logo, é possível que ele possa vir a ser configurado para tirar proveito da redução de trocas de contexto, por exemplo, aumentando o parâmetro *sched_min_granularity* para uma fatia de tempo mínima bem maior que o default do equipamento em questão.

Grupo com 2, 3 ou 4 processos concorrentes de 10 segundos de **CF sem *CFS***

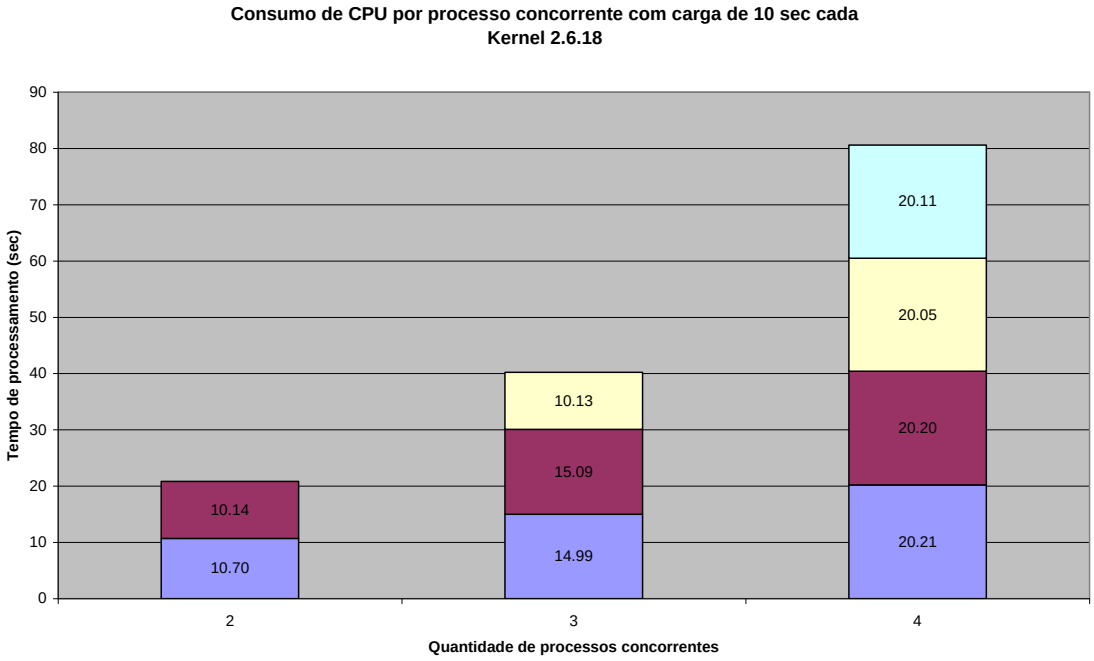

Figura 3-5 Gráfico do consumo de CPU por processo concorrente. Sem *CFS*.

Grupo com 2, 3 ou 4 processos concorrentes de 10 segundos de **CF com *CFS***

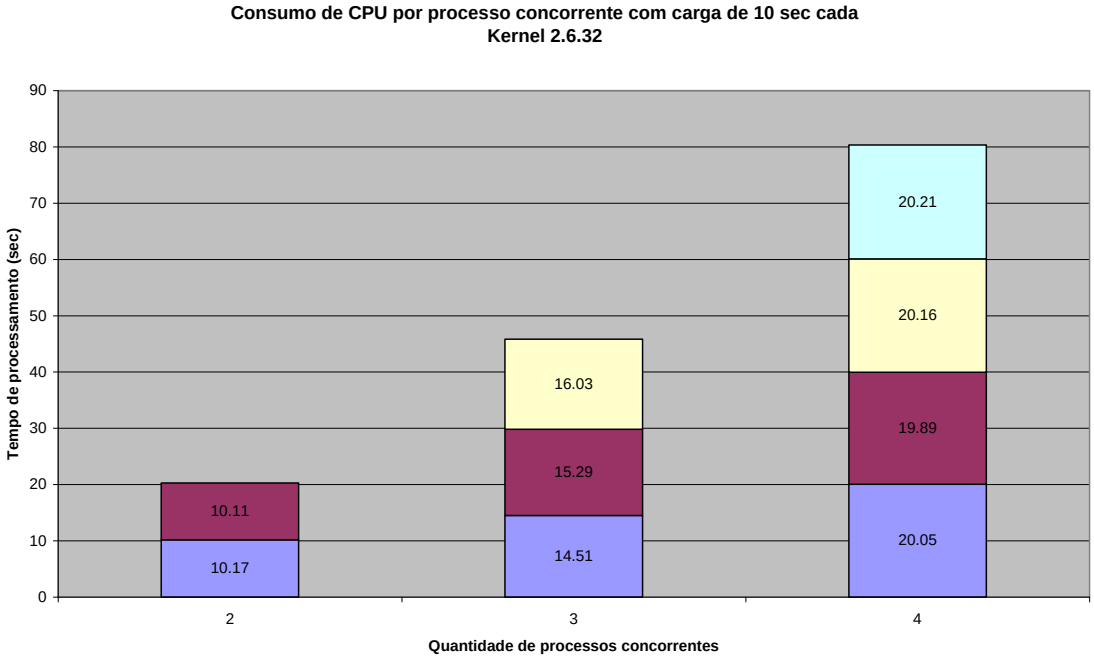

Figura 3-6 Gráfico do consumo de CPU por processo concorrente. Com *CFS*.

Para o caso de 2, 3 ou 4 processos de 10 segundos cada, representados nas Figuras 3-5 e 3-6, vale o mesmo raciocínio do caso correspondente a 1 segundo, isto é, para um número de processos múltiplo do número de *cores*, os tempos de CPU serão próximos para cada processo, enquanto para as variâncias temos:

CFS	Número de Processadores		
	2	3	4
sem	0,160813	8,039991	0,005795
com	0,001711	0,580772	0,020065

Tabela 3-3 Variâncias em pequena escala com CF = 10 segundos

Novamente, as variâncias revelam que o escalonador **com *CFS*,** pode ser menos injusto que o escalonador **sem *CFS***, mas a um alto custo. Se a justiça compensa um baixo desempenho, depende dos administradores do sistema e das necessidades das aplicações. Deve-se notar a possibilidade do ***CFS*** ser reconfigurável dinamicamente.

Grupo com 5, 7 ou 10 processos concorrentes de 10 segundos de **CF sem *CFS***

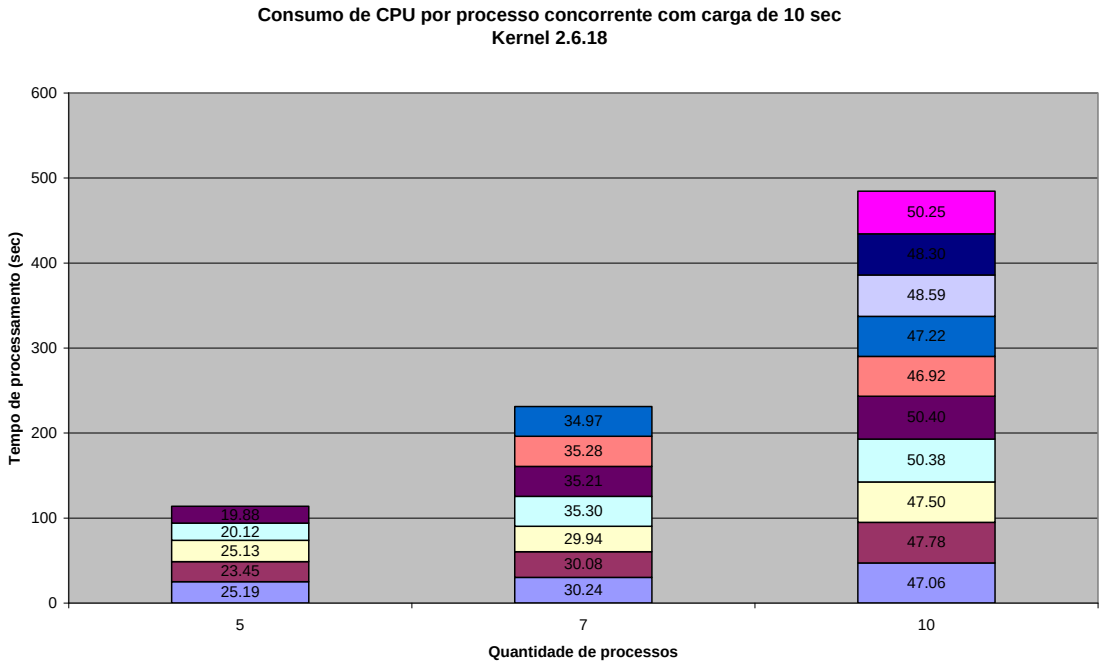

Figura 3-7 Gráfico do consumo de CPU por processo concorrente. Sem *CFS*.

Grupo com 5, 7 ou 10 processos concorrentes de 10 segundos de **CF com *CFS***

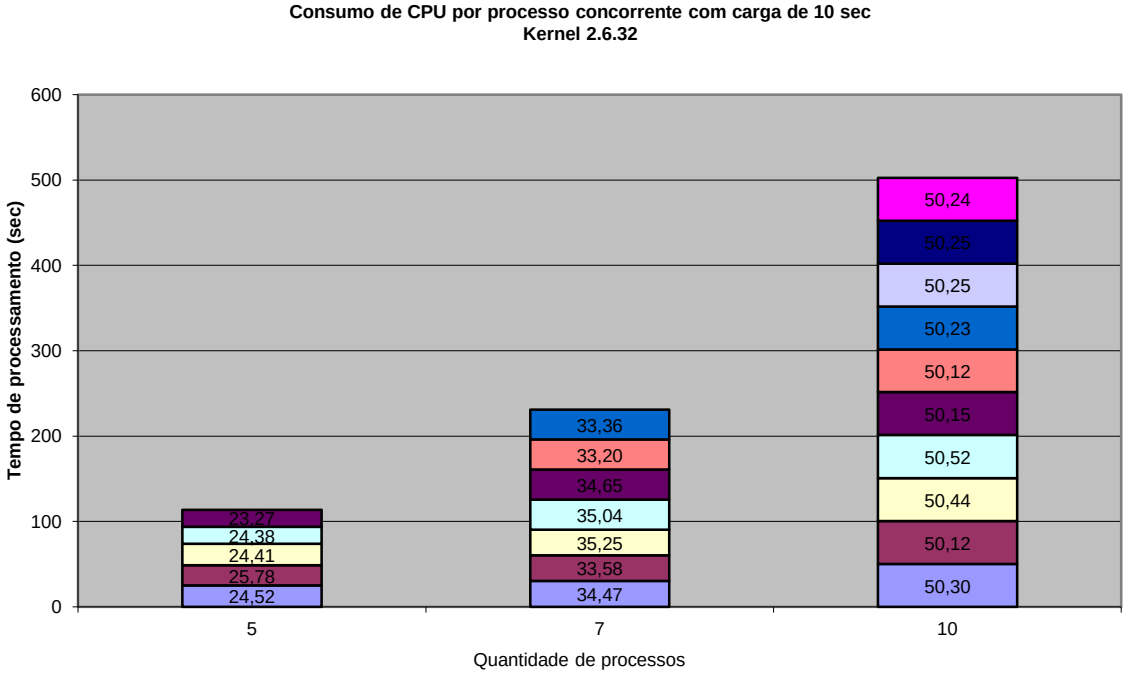

Figura 3-8 Gráfico do consumo de CPU por processo concorrente. Com *CFS*.

Para o caso do aumento de processos e da carga de cada processo, descritos nas Figuras 3-7 e 3-8, identificamos que o ***CFS*** mostra-se claramente menos injusto que seu antecessor pois, tudo indica que parece valer a pena distribuir fatias menores de tempo mais vezes, com o ônus da troca de contexto (os módulos de carga, no caso são mínimos, da ordem de 4K) pois o escalonador tende a ser mais justo como se propõe.

Para as variâncias temos:

CFS	Número de Processadores		
	5	7	10
sem	6,811728	7,471757	1,990584
com	0,791426	0,729237	0,017505

Tabela 3-4 Variâncias em pequena escala com CF = 10 segundos

Novamente, as variâncias revelam claramente que o escalonador **com *CFS*** aparenta ser menos injusto em dois casos (5 e 7 processos). Note-se que os valores das variâncias para o ***CFS*** são muito próximos porém mais baixos que os do **sem *CFS***.

Os estudos dos gráficos correspondentes às Figuras 3-1 a 3-8, referentes a processadores com dois núcleos, estão sintetizados na Tabela 3-5.

# Processos concorrentes	Média de tempo com CFS		Média de tempo sem CFS	
	1 seg	10 seg	1 seg	10 seg
2	1,02	10,14	1,03	10,42
3	1,42	15,28	1,37	13,40
4	2,01	20,08	1,94	20,15
5	2,38	24,47	2,24	22,75
7	3,23	34,25	3,29	33,00
10	4,73	50,26	4,70	48,44
# Processos concorrentes	**Variância com CFS**		**Variância sem CFS**	
	1 seg	10 seg	1 seg	10 seg
2	0,00	0,00	0,00	0,16
3	0,03	0,58	0,08	8,04
4	0,00	0,02	0,00	0,01
5	0,08	0,79	0,07	6,81
7	0,05	0,73	0,04	7,47
10	0,10	0,02	0,03	1,99

Tabela 3-5 Resultados consolidados relativos às Figuras 3-1 a 3-8

A Tabela 3-5 resume os resultados experimentais preliminares do ponto de vista quantitativo. A primeira coluna relaciona o número de processos concorrentes; os valores médios **com** e **sem *CFS*** são mostrados nas colunas 2 e 3, respectivamente para 1 e 10 segundos de **CF**; nas duas últimas colunas, 4 e 5, estão as média **sem *CFS***. Na parte inferior da Tabela 3-5, no lugar das médias encontram-se as variâncias dos tempos de CPU de cada execução **com** e **sem** o ***CFS***.

Da Tabela 3-5, conclui-se que, a partir da análise das variâncias dos dados que o ***CFS*** é menos injusto no caso de poucas CPUs (*dual core*) de maneira significativa – as variâncias relativas a cada escalonador são distintas e elevadas para o *kernel* **sem *CFS*** em relação ao **com *CFS***. Esta justiça é obtida até aqui a um alto custo de CPU. Portanto, pelo pequeno número de amostras, ainda não se pode afirmar, que o escalonador **com *CFS*** é mais justo do que o **sem *CFS***. Certamente, são necessários mais experimentos, como os que serão realizados a seguir.

Os dados dos gráficos indicam que o escalonador CFS, em geral, é menos injusto que seu antecessor, ***O(1)*** a um alto custo, devido possivelmente a elevada troca de contexto, como verificaremos mais adiante. Dado que o ***CFS*** é dinamicamente reconfigurável, buscaremos reduzir esta excessiva troca de contexto reparametrizando o *kernel* no sentido de aumentar a latência mínima e granularidade mínima do escalonador ***CFS***. Todos os dados gerados para o *kernel* 2.6.32 já com o ***CFS*** inserido no próprio *kernel*, foram para os seguintes parâmetros *default* do *kernel*:

kernel.sched_latency_ns = 20000000 (20 ms)

kernel.sched_min_granularity_ns = 4000000 (4 ms)

Estas considerações serão relevantes para uma nova parametrização a ser proposta e realizada com os mesmos dados de número de processos e *quanta* de tempos nos capítulos seguintes, como uma tentativa de melhoria dos *jobs CPU-bound*.

Em todos os gráficos anteriores, obtidos com processador Intel Core2 em *notebooks* Sony VAIO e DELL M6400, podemos constatar visualmente através dos gráficos das Figuras 3-1 a 3-8 e, quantitativamente através da análise das variâncias associadas que o tempo de processamento de cada processo aumenta proporcionalmente ao número de processos, logicamente, mas tende a se aproximar quando o número de processos cresce **com** ou **sem** o ***CFS***. Neste ponto, enfrenta-se uma influência devida a

impossibilidade de submeter os processos concorrentes de modo síncrono. Isto significa que alguns processos terminarão bem antes que outros, mas como estamos interessados no comportamento quantitativo médio do consumo de CPU dos processos, este efeito será desconsiderado para fins práticos.

3.3.2 Balanceamento de carga (*underprovisioning*)

Neste experimento, o objetivo é verificar o comportamento do escalonador ***CFS*** em relação ao balanceamento de carga com o aumento do número de processos concorrentes em cada instante de tempo.

Os dados da Tabela 3-6 foram obtidos a partir da submissão de 12 processos concorrentes com o método **LI** de consumo de CPU. Pode-se identificar que o mecanismo básico de ativação de cada CPU e a distribuição de carga através do escalonador ***CFS*** satisfazem a um algoritmo do tipo *round-robin* e funcionam muito bem segundo um equilibrado balanceamento de carga entre as diversas CPUs do sistema. Eles foram obtidos em uma estação de trabalho HP Z800 com 2 processadores Intel Xeon 5680 *hexacore* de 3.2 GHz e 12 MB de *cache*.

T (seg)	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7	CPU 8	CPU 9	CPU 10	CPU 11	CPU Total (*mseg*)
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1			1000										1000
2			1000				1000						2000
3				1000			1000	1000					3000
4	1000			1000			1000	1000					4000
5	535	822	1000				642	1000			1000		5000
6	995	1000		1000			5	1000		1000	1000		6000
7	840	1000	994	1000			160	7	1	1000	1000	1000	7002
8	992	1000	1000	1000			1000		10	1000	1000	1000	8004
9	42	1000	1000	1000	995	1	88	1000	875	1001	1000	1000	9003
10	996	1000	1000	1000	1000	2	6	1000	1000	1000	1000	1000	10004
11	627	1000	1000	1000	1000	878	499	1000	1000	1000	1000	1000	11005
12	1004	1004	1004	1004	1004	1003	1002	1004	1004	1003	1004	1004	12041

Tabela 3-6 Distribuição de carga de CPU por *core* em um instante T

A coluna 1 da Tabela 3-6, representa o passo de tempo (T), a partir de 0 (zero) e incrementado de uma unidade a cada linha. As colunas seguintes representam o tempo

de processamento acumulado de cada CPU naquele instante T desde o instante inicial 0. No tempo T=0, os tempos de processamento de todas as CPUs estão zerados. Na última coluna, temos o tempo total de CPU acumulado, em milissegundos, da soma dos tempos de processamento de todas as CPUs desde T=0.

Para cada instante T, de 1 em 1 segundo, é criado um processo **LI**. Em cada segundo subsequente, 1000 unidades de tempo de CPU, dados em milissegundos (por conveniência dos resultados contabilizados pelo comando *procstat),* são alocados para cada *core* à medida que os processos deles necessitem, comprovando que o escalonador utiliza muito bem os sistemas *multicore*. Os *cores* identificados pelas CPUs de 0 a 5 pertencem ao processador físico 1 (*package* 1) e os *cores* seguintes, de 6 a 11, ao processador físico 2 (*package* 2). O balanceamento é ideal, no caso, do ponto de vista da distribuição espacial, pois existem dois processadores físicos *hexacore* e os processos vão sendo distribuídos entre eles equitativamente. Embora exista uma otimização espacial, que no caso específico não é ótimo do ponto de vista do consumo energético, porque obriga que os dois processadores estejam consumindo energia ao mesmo tempo. Talvez seja mais conveniente deixar "ativo" apenas um processador, até que todos os *cores* estejam trabalhando e então, seja preciso mais um. Isto poderia refletir apenas um balanceamento energético no sistema como um todo.

3.3.3 Impacto da reconfiguração dinâmica do *CFS*

Nos experimentos anteriores, foi verificado que o ***CFS*** tem uma certa sobrecarga com trocas de contexto. O próximo experimento visa analisar se é possível melhorar o desempenho do ***CFS*** com a reparametrização do escalonador.

Reparametrizando o tamanho de sua época ou latência de escalonamento e sua granularidade mínima, dadas por seus parâmetros dinamicamente reconfiguráveis de *kernel* : *kernel.sched_latency_ns* e *kernel.sched_min_granularity_ns* podemos adaptar o escalonador ***CFS*** no sentido de obter um melhor tempo real de resposta de cada processo. Esta reparametrização é conveniente principalmente quando sabemos, a priori, a carga de cada processo e se estas cargas são aproximadamente idênticas.

Os experimentos apresentados nos gráficos das Figuras 3-9 e 3-10, a seguir, foram realizados um *notebook* HP Elitebook com processador Intel i7 *quadcore* QM2870 de 2.3 GHz de *clock*. O objetivo é tentar reduzir o número de trocas de contexto através da parametrização do escalonador de modo a aumentar o trabalho útil e a eficiência de aplicações voltadas para uso intensivo de CPU, por exemplo.

Para 4 processos de 1 segundo de **CF** com parâmetros *default* do escalonador ***CFS*** **(**latência = 20ms e granularidade mínima = 5ms)

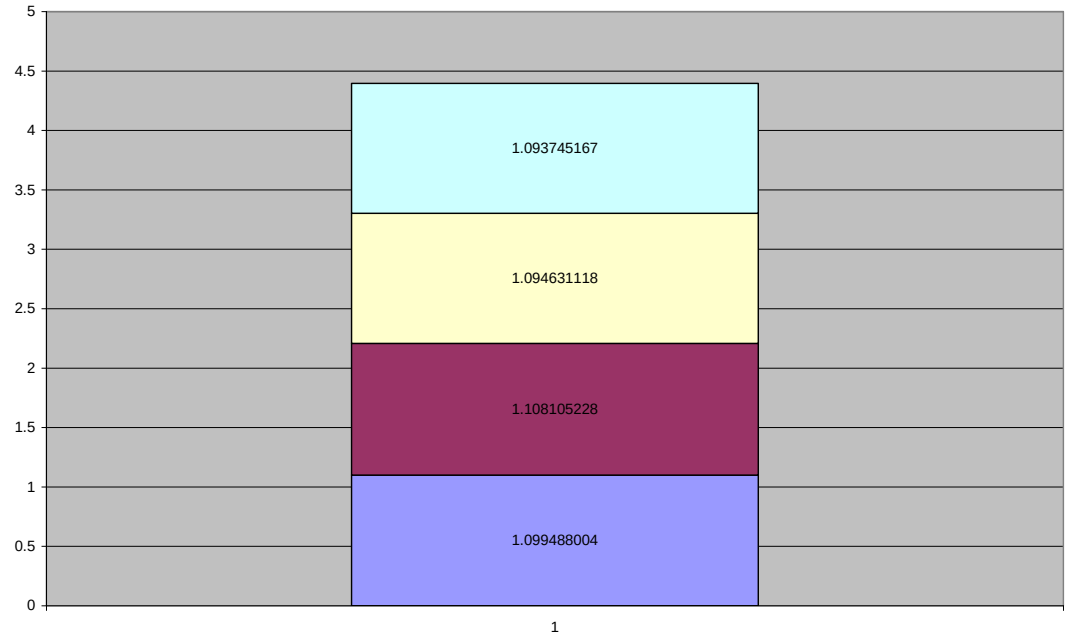

Figura 3-9 Gráfico do tempo de CPU para 4 processos de 1 segundo.

Para 4 processos de 1 segundo de **CF** com parâmetros modificados do escalonador ***CFS*** **(** latência = 400ms e granularidade mínima = 100ms)

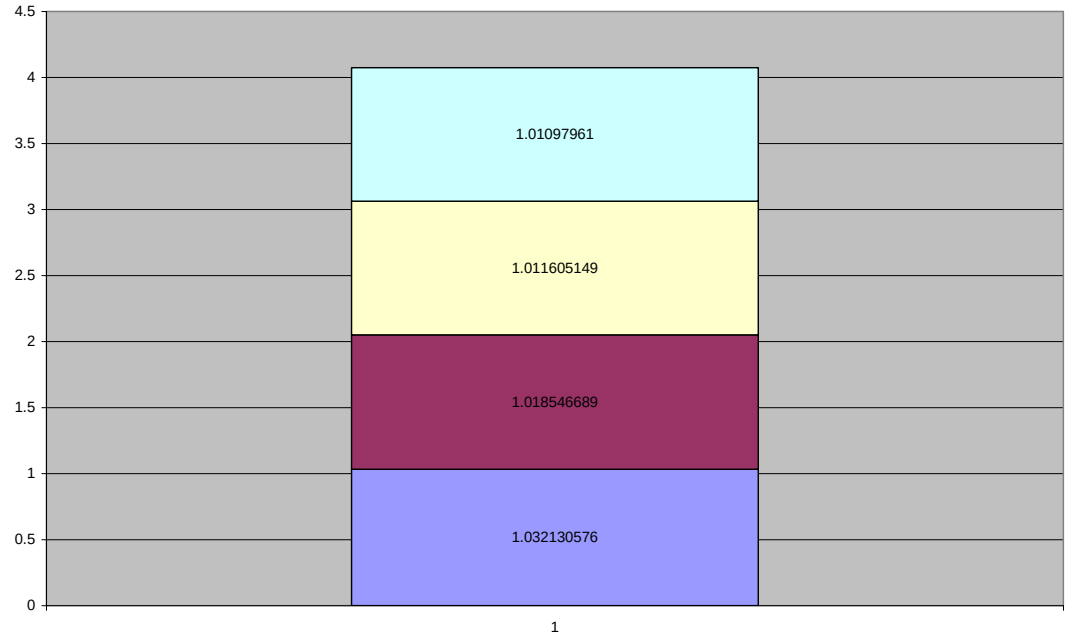

Figura 3-10 Gráfico de tempo de CPU para 4 processos de 1 segundo.

Os dois gráficos das Figuras 3-9 e 3-10 foram obtidos com diferentes parâmetros para a latência e granularidade mínimas do escalonador ***CFS*** para quatro processos concorrentes. O primeiro foi com o *default*: *20 ms e 5 ms*. O segundo, com o padrão modificado para *400 ms* e *100 ms*, isto é, vinte vezes mais que o *default*. O resultado, como pode ser visto no gráfico da Figura 3-10, é de uma significativa redução do tempo total de processamento, aproximadamente 9%, sem nenhum prejuízo para nenhum processo em questão. Isto implica que podemos dinamicamente reparametrizar o escalonador ***CFS*** para melhorar o tempo de resposta final de processos CPU-*bound* com significativos ganhos de desempenho, mas com um ônus no tempo de resposta se os processos forem interativos.

3.4 RESUMO

Neste capítulo foi realizado o detalhamento e síntese do ***CFS*** e apresentado seu algoritmo fundamental assim como seus principais e inovadores conceitos e funcionalidades gerais. São apresentados os métodos de abordagem **LI** e **CF** para uso nos testes e os primeiros experimentos para familiarização com o comportamento do escalonador e a comparação inicial com seu antecessor em pequena escala.

A pequena escala não permite ainda generalizações a respeito do ***CFS***. O escalonador ***CFS*** busca a justiça sacrificando o desempenho com o aumento da troca de contexto [81] [82]. Dada a possibilidade de reconfiguração dinâmica do ***CFS***, através da reparametrização de parâmetros críticos para o ***CFS*** como o *sched_latency_ns*, *sched_min_granularity_ns* e *sched_min_wakeup_ns* podemos adequar o escalonador dinamicamente para obter uma minimização de troca de contexto para melhorar aplicações do tipo CPU intensivas, por exemplo, que são fundamentais nos ambientes de HPC e Nuvem [83].

No capítulo seguinte, serão testados experimentos em grande escala (muitas CPUs, processos e tempo de amostragem) objetivando buscar uma visão de possíveis padrões do ***CFS*** para muitas CPUs e processos concorrentes no sistema para melhorar a análise do escalonador ***CFS***.

CAPÍTULO 4 – ANÁLISE EXPERIMENTAL

Neste capítulo examinaremos de maneira mais exaustiva, detalhada e refinada, os resultados experimentais com a presença ou não, em alguns casos, do escalonador atual ***CFS*** ou de seu antecessor, ***O(1)***, anterior ao *kernel* Linux 2.6.23. Os experimentos utilizam as aplicações **LI** e/ou **CF,** descritos anteriormente na seção 3.2. Os resultados finais serão apresentados em gráficos e/ou tabelas para melhor esclarecimento.

4.1 BALANCEAMENTO DE CARGA (*UNDERPROVISIONING)*

O balanceamento de carga de um escalonador é bem definido e equilibrado se o escalonador divide a carga de trabalho de processamento dentre as CPUs de maneira igualitária por unidade de tempo. Em outras palavras, o escalonador deve buscar carregar todas as CPUs existentes e disponíveis de modo tal que cada uma delas processe as tarefas cedendo quantidade de carga de processamento praticamente uniforme por unidade de tempo.

No experimento apresentado nesta seção, é avaliado se o escalonador ***CFS*** é balanceado em relação aos processadores para uma quantidade de processos maior que o número de CPUs existente (*overprovisioning*).

A aplicação **LI** é usada em um equipamento com quatro *cores* ou CPUs. Foram efetuadas várias execuções desses *loops* com número de processos ou tarefas concorrentes variando de 1 a 12. Os resultados podem ser visualizados na Tabela 4-1. Nesta tabela, as primeiras quatro colunas correspondem aos tempos de processamento de cada CPU à medida que é criado um novo processo do tipo **LI** a cada segundo, representado por cada linha da tabela. A última coluna tem o tempo cumulativo de todas as CPUs a cada segundo, desde o instante inicial até o fim de cada teste. No início de cada teste a linha de tempo de CPU inicial é zerado.

Tempo de CPU (x 0.01 s)				
cpu0	cpu1	cpu2	cpu3	Total CPU
1 Processo				
0	0	0	0	0
100	1	1	0	102
2 Processos				
0	0	0	0	0
0	100	1	0	101
0	200	1	100	301
3 Processos				
0	0	0	0	0
0	1	100	0	101
100	1	200	0	301
200	1	300	100	602
4 Processos				
0	0	0	0	0
1	100	1	0	101
1	200	1	100	301
1	300	101	200	601
103	403	204	303	1012
5 Processos				
0	0	0	0	0
1	101	1	0	101
1	201	101	0	301
1	301	201	100	601
103	403	304	203	1012
206	506	406	305	1421
6 Processos				
0	0	0	0	0
0	100	1	0	101
0	201	101	0	301
0	301	201	100	602
102	403	303	202	1010
204	505	405	304	1418
307	608	508	407	1830
7 Processos				
0	0	0	0	0
0	1	0	100	102
0	1	100	200	302
0	101	200	300	602
103	204	303	403	1013
206	306	406	505	1424
308	409	508	607	1832

Tempo de CPU (x 0.01 s)				
cpu0	cpu0	cpu0	cpu0	cpu0
410	511	610	709	2240
8 Processos				
0	0	0	0	0
1	100	0	0	101
1	200	0	100	301
1	300	100	200	602
103	403	203	303	1012
206	505	305	406	1422
309	607	408	508	1832
410	709	510	610	2240
514	813	613	713	2654
9 Processos				
0	0	0	0	0
1	100	1	0	101
1	200	1	100	301
1	300	101	201	602
103	403	204	303	1013
206	505	306	406	1422
308	608	409	508	1833
410	709	510	610	2238
513	813	614	713	2652
616	916	717	816	3065
10 Processos				
0	0	0	0	0
0	100	0	1	102
0	200	100	1	302
0	300	200	101	602
103	403	303	203	1012
205	506	406	306	1423
308	608	508	408	1834
409	710	610	510	2239
512	813	713	613	2651
615	916	816	716	3062
717	1018	918	818	3471
11 Processos				
0	0	0	0	0
0	1	100	0	101
0	101	200	0	301
0	201	300	100	602
103	304	403	203	1013
205	407	506	306	1423
308	509	607	408	1832
410	611	709	510	2240
512	714	812	613	2650

Tempo de CPU (x 0.01 s)				
cpu0	cpu0	cpu0	cpu0	cpu0
619	821	919	719	3078
722	923	1021	822	3488
825	1027	1125	925	3902
12 Processos				
0	0	0	0	0
0	1	100	0	101
100	1	200	0	301
201	101	301	0	601
303	203	403	102	1010
406	306	505	205	1420
508	408	608	307	1830
610	510	710	409	2238
712	613	812	511	2648
816	716	916	615	3061
922	822	1022	721	3485
1024	924	1124	823	3894
1127	1027	1227	926	4306

Tabela 4-1 Distribuição da carga de CPU por *core* no tempo com até 12 processos concorrentes executando em 4 *cores*

Analisando os dados, verifica-se que o escalonador ***CFS***, à medida que um novo processo entra na fila de execução, procura oferecer uma CPU ociosa, se existir, para este novo processo mesmo que seja um novo processo que venha a consumir um tempo ínfimo de processamento, como os de monitoração, por exemplo. Isto justifica os valores apresentados de 1, 2 ou poucos centésimos de segundo, logicamente inclusos nos tempos gastos pelo processador. O escalonador busca distribuir de modo igualitário a carga para cada CPU, mas não necessariamente o tempo de cada processo, mas sim de todos eles como um todo, como veremos mais a frente.

Os pequenos consumos de CPU (da ordem de poucos milissegundos) são devidos ao custo da monitoração que também concorre pelas quatro CPUs, que para o escalonador é mais um novo processo a concorrer pelos recursos, logicamente. Os processos **LI** concorrem com os de monitoração pelas CPUs do sistema. Os valores de consumo ligeiramente maiores que 100 unidades (em centésimos de segundos para compatibilizar com a taxa de amostragem da versão do utilitário *procstat* instalado no Linux do *quadcore*) também são devidos à defasagem da atuação da monitoração e impedem necessariamente a precisão demarcada dos intervalos de tempo de monitoração de 1 segundo. O erro das medidas é de 1 ms, embora os tempos no ***CFS***

sejam mensurados em nanossegundos. Esta diferença deve-se ao fato de compatibilização da ferramenta de monitoração estatística, *procstat*, por compatibilidade desta com as versões anteriores do escalonador. Mas, os erros associados são sempre de nanossegundos neste trabalho.

Outra conclusão relevante é que a carga total do sistema é máxima e aproximadamente 400 unidades de CPU (x 0,01) e praticamente se preserva no tempo à medida que o número de processos CPU-intensivos aumenta. Isto é, o escalonador é extremamente eficiente para processos que consomem muita CPU por ser leve, ou seja, decidir sem muito custo o próximo processo a executar devido a sua heurística muito simples e sempre buscar dividir o tempo de processamento de todas as CPUs por todos os processos de modo equânime. Por outro lado, o custo de gerenciamento também vai aumentando sutilmente com a carga do sistema, pois o próprio gerenciamento também compete a cada intervalo de amostragem com cada vez mais e mais processos concorrentes.

4.2 BALANCEAMENTO DE CARGA – EFEITO *PING-PONG*

Neste experimento, testa-se a persistência ou continuidade do processo em uma mesma CPU, evitando-se assim o efeito *ping-pong* ou ineficiência do escalonador. O teste consiste em submeter processos da aplicação **CF** com carga fixa de 10 segundos e com novo processo idêntico a cada novo intervalo de 10 segundos. Os dados correspondentes a cada coluna de CPU são cumulativos para cada CPU de 10 em 10 segundos. Os resultados são apresentados na Tabela 4-2.

Caso exista um número menor de processos do que de CPUs, o escalonador procura manter o processos na mesma CPU para evitar trocas de contexto ou outras perdas como trocas de CPU de mesmo ***chip*** para um outro distinto ***chip***, como pode ser verificado na Tabela 4-2, quando é feita a submissão de um único processo de 10 segundos.

Observa-se nos resultados apresentados, que o balanceador de carga do ***CFS***, ser neste caso, procura apenas distribuir de maneira equilibrada a carga de processamento entre as CPUs. Em outros casos, com múltiplos processadores físicos (*packages*), também pode ser conveniente procurar dividir a carga de processamento visando um certo balanceamento de energia. Isto significa uma tentativa de minimizar o consumo de energia fazendo com que o menor número de CPUs físicas necessárias para a carga de processamento seja utilizado, tentando-se usar ao máximo o maior número de *cores* por

CPU ao invés de uma distribuição esparsa dentre as CPUs físicas. Existem parâmetros de configuração do escalonador para tal.

Tempo de CPU (x 0,01 segundo)				
cpu0	cpu1	cpu2	cpu3	Total CPU
1 Processo de 10 segundos de CPU				
0	0	0	0	0
1	1	0	1004	1006
2 Processos de 10 segundos de CPU				
0	0	0	0	0
1000	1	1	0	1002
1000	1	1001	0	2002
3 Processos de 10 segundos de CPU				
0	0	0	0	0
1	1	0	1000	1002
0	1000	1	1000	2001
1001	1001	0	1002	3004
4 Processos de 10 segundos de CPU				
0	0	0	0	0
1	1	0	1000	1002
0	0	1000	999	1999
1001	2	1002	1002	3007
1003	988	1002	1002	3995

Tabela 4-2 Preservação e manutenção do processo no próprio *core* p/ 4 *cores*

Pela análise dos dados da Tabela 4-2, observa-se que mesmo para cargas fixas (**CF**) de 10 segundos não existe o efeito *ping-pong* da aplicação ou processo saltitando de *core* em *core* desnecessariamente. O escalonador procura manter ao máximo o processo em sua CPU afim.

4.3 AUTO-CONFIGURAÇÃO DE PARÂMETROS

O experimento a seguir é útil para demonstrar que o escalonador ***CFS*** se autoconfigura dinamicamente. Portanto, o *kernel* adapta-se dinamicamente às alterações que impliquem em adição ou subtração de recursos como CPUs.

O experimento consiste em alterar dinamicamente o número de CPUs ativas de 1 (valor mínimo obrigatório) até o máximo (número de *cores* ou número máximo de *threads* para CPUs com tecnologia HT ou SMT habilitada). Para cada caso, são listadas as variáveis de ambiente do escalonador ***CFS*** representadas nas colunas de 4 a 6 da

Tabela 4-3. Os dados são tabulados na tabela 4-3, onde a coluna representa a plataforma de hardware; a coluna 2 a CPU ou *core* ativo; e a 3 coluna se o HT ou SMT está ligado (*on*) ou desligado (*off*).

Plataforma de Hardware	CPU Ativa	CPU c/ SMT on/off	min_latency_ns (ns)	min_granularity_ns (ns)	wakeup_granularity_ns(ns)
VAIO Dual Core	1	NA	5000000	1000000	1000000
	2	NA	10000000	2000000	2000000
HP Elitebook Quad Core	1	*off*	20000000	4000000	4000000
	2	*off*	20000000	4000000	4000000
	3	*off*	20000000	4000000	4000000
	4	*off*	15000000	3000000	3000000
	5	*on*	15000000	3000000	3000000
	6	*on*	15000000	3000000	3000000
	7	*on*	15000000	3000000	3000000
	8	*on*	10000000	2000000	2000000
HP Z800 2 CPU - 12 *cores*	1	*off*	10000000	2000000	2000000
	2	*off*	10000000	2000000	2000000
	3	*off*	15000000	3000000	3000000
	4	*off*	15000000	3000000	3000000
	5	*off*	15000000	3000000	3000000
	6	*off*	15000000	3000000	3000000
	7	*off*	20000000	4000000	4000000
	8	*off*	20000000	4000000	4000000
	9	*off*	20000000	4000000	4000000
	10	*off*	20000000	4000000	4000000
	11	*off*	20000000	4000000	4000000
	12	*off*	20000000	4000000	4000000
HP Z820 2 CPU - 16 *cores*	1	*off*	12000000	150000	2000000
	2	*off*	12000000	150000	2000000
	3	*off*	18000000	225000	3000000
	4	*off*	18000000	225000	3000000
	5	*off*	18000000	225000	3000000
	6	*off*	24000000	3000000	4000000
	7	*off*	24000000	3000000	4000000
	8	*off*	24000000	3000000	4000000
	9	*off*	24000000	3000000	4000000
	10	*off*	24000000	3000000	4000000
	11	*off*	24000000	3000000	4000000
	12	*off*	24000000	3000000	4000000
	13	*off*	24000000	3000000	4000000
	14	*off*	24000000	3000000	4000000
	15	*off*	24000000	3000000	4000000
	16	*off*	24000000	3000000	4000000

Tabela 4-3 Variação dinâmica de parâmetros do *CFS*

Por exemplo, o *notebook* VAIO *Dual Core* sem tecnologia *HyperThread* (HT) possui, por *default*, os parâmetros *min_latency_ns* = 5000000, *min_granularity_ns* =

1000000 e *wakeup_granularity_ns* = 1000000 com apenas uma CPU ativa. Quando a segunda CPU é ativa estes parâmetros são automaticamente alterados para *min_latency_ns* = 10000000, *min_granularity_ns* = 2000000 e *wakeup_granularity_ns* = 2000000. Estes novos parâmetros são aumentados para maior escalabilidade do sistema. Vale o raciocínio contrário, se recursos são retirados por alguma razão, os valores são reduzidos para preservação da justiça do escalonador ***CFS***.

Da Tabela 4.3, conclui-se que à medida que os recursos são reduzidos, ou seja, as CPUs são inibidas ou desligadas, o escalonador ***CFS*** se autoparametriza, como pode ser visualizado nos valores da tabela através das colunas 2, 4, 5 e 6. Logo, o *kernel* de fato se autoconfigura dinamicamente, modificando parâmetros de seu escalonador ***CFS*** ao detectar retirada ou subtração de recursos, como CPU, por exemplo.

4.4 AVALIANDO A EQUIDADE DO *CFS* EM LARGA ESCALA – VARIANDO O NÚMERO DE PROCESSOS

Este experimento consiste em verificar a equidade ou justiça do escalonador ***CFS*** em larga escala, isto é, muitos processadores e muitos processos. O objetivo é avaliar a identificação de padrões de comportamento do escalonador. Para tal, serão submetidos jobs concorrentes em *background* em uma *workstation* com dois processadores físicos *hexacore*, em um total de 12 *cores*, tal que cada processo busque o máximo de CPU que lhe puder ser cedida, através da aplicação **LI**. Acrescenta-se a cada intervalo de tempo determinado de 60 segundos um processo a mais. Monitora-se o consumo de CPU por cada processo per si a cada intervalo de 60 segundos. No final, os dados são tabulados e, a partir deles, são obtidos gráficos de compartilhamento percentual, correspondentes a cada intervalo de tempo de 60 segundos. Estes experimentos são testados para cada *kernel*, **com** e **sem** o novo escalonador ***CFS***.

Inicialmente, foram submetidos 48 processos concorrentes, disparando um a um a cada intervalo de tempo de 60 segundos com a estratégia **LI,** para sistemas **sem** ou **com** o ***CFS,*** respectivamente, e, após o cálculo dos tempos de CPU consumidos por cada processo naquele intervalo de tempo de 60 segundos correspondente. A escolha da quantidade de 48 processos é para que se tenha pelo menos 4 processos por CPU, em média. Logo a seguir, trabalharemos com 100 processos concorrentes para fins de larga escala e verificação de padrões.

Os seguintes gráficos, representados nas Figuras 4-1 e 4-2, foram obtidos:

48 processos – um novo a cada 60 segundos **sem *CFS***

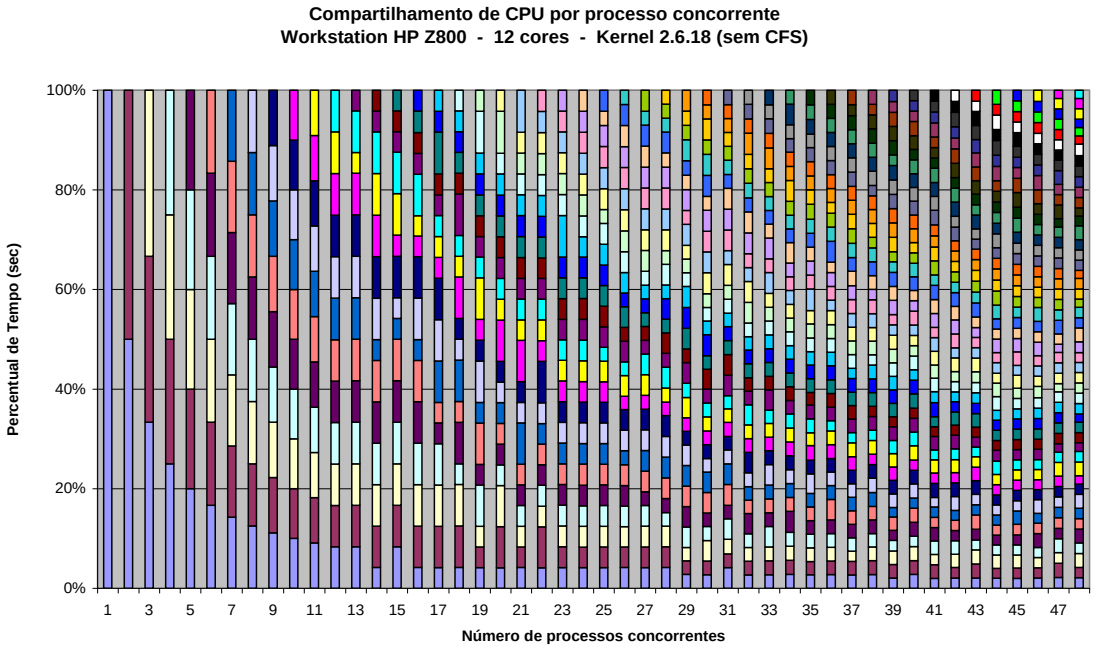

Figura 4-1 Compartilhamento de CPU por até 48 processos concorrentes - 1 novo processo é disparado a cada 60 segundos sem *CFS*

48 processos – um novo a cada 60 segundos **com *CFS***

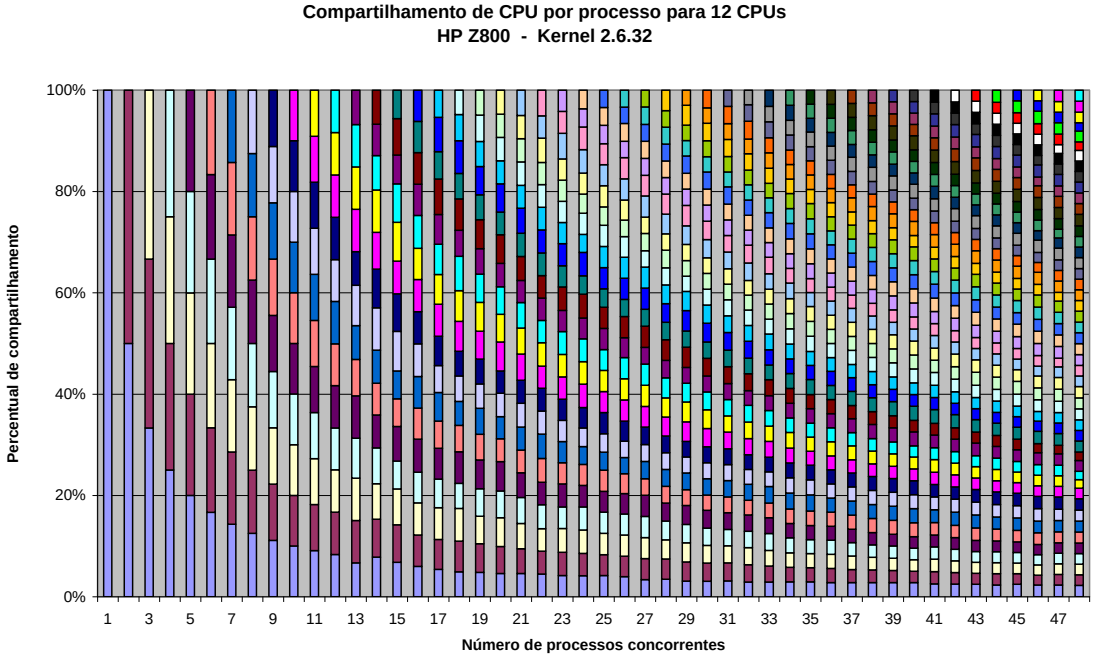

Figura 4-2 Compartilhamento de CPU por até 48 processos concorrentes - 1 novo processo é disparado a cada 60 segundos com *CFS*

A análise qualitativa das Figura 4-1 (**sem *CFS***) e Figura 4-2 (**com *CFS***) revelam o seguinte para 48 processos de 60 segundos cada:

Nos gráficos das duas Figuras 4-1 e 4-2, o comportamento da distribuição ou compartilhamento de CPU pelos 12 processadores é graficamente o mesmo até a saturação mínima correspondente a 12 processos, um para cada CPU. À medida que o número de processos aumenta, nitidamente (a ser demonstrado quantitativamente através da análise das variâncias) o *kernel* **sem *CFS*** aparenta distribuir injustamente os tempos de CPU dentre os processos concorrentes que disputam os recursos do sistema.

O gráfico da Figura 4-1, correspondente ao *kernel* **sem *CFS***, aparenta tornar-se mais irregular. A irregularidade exibe a iniquidade à medida que o número de processos aumenta. O comportamento irregular da distribuição de CPU pelos processos, representada pelos tamanhos dos segmentos coloridos, e mais facilmente identificável próximo a múltiplos do número de *cores*, quando as barras coloridas passam a representar valores desproporcionais em cada intervalo de tempo, ou seja, diferentes de 60 segundos, no caso específico.

O gráfico da Figura 4-2, correspondente ao *kernel* **com *CFS***, já aparenta uma melhor e mais equitativa distribuição do tempo de CPU dentre os processos, pois aparentam um queda suave e visualmente valores regulares e proporcionais ao número de processos concorrentes independentemente do número de processos.

As variâncias para processos concorrentes de 13 a 21 (eixo dos X) com 48 processos de 60 segundos cada são mostradas a seguir.

Da Figura 4-1 (**sem *CFS***)

13	14	15	16	17	18	19	20	21
128,1246	201,9168	236,5115	245,8951	239,9896	213,2898	187,9387	152,3271	121,1045

Da Figura 4-2 (**com *CFS***)

13	14	15	16	17	18	19	20	21
23,6336	12,3792	32,7204	3,3811	8,1855	1,8092	6,2074	1,0013	4,0870

Para a variância populacional[7], encontramos os valores:

sem *CFS*	190,6300
com *CFS*	159,8417

As variâncias bem menores para as amostras do Figura 4-2, **com *CFS***, em relação as amostras da Figura 4-1, **sem *CFS***, indicam que os tempos são distribuídos mais equanimemente pelo ***CFS***.

O mesmo experimento foi realizado considerando um número maior de processos, como pode ser visto a seguir.

[7] **Variância populacional** é a variância de todas as amostras do experimento (população). É dada pelo quadrado do desvio padrão de todas as amostras ($var = \delta^2 = \sum_{i=1}^{N}(x_i - \bar{x})^2$). Onde,
N é o número total de amostragens; x_i é o valor de cada amostra; $\bar{x}$ é o valor esperado da amostra; i=(1,2,...,N); e, δ é o desvio padrão.

100 processos a cada 60 segundos **sem** o ***CFS***

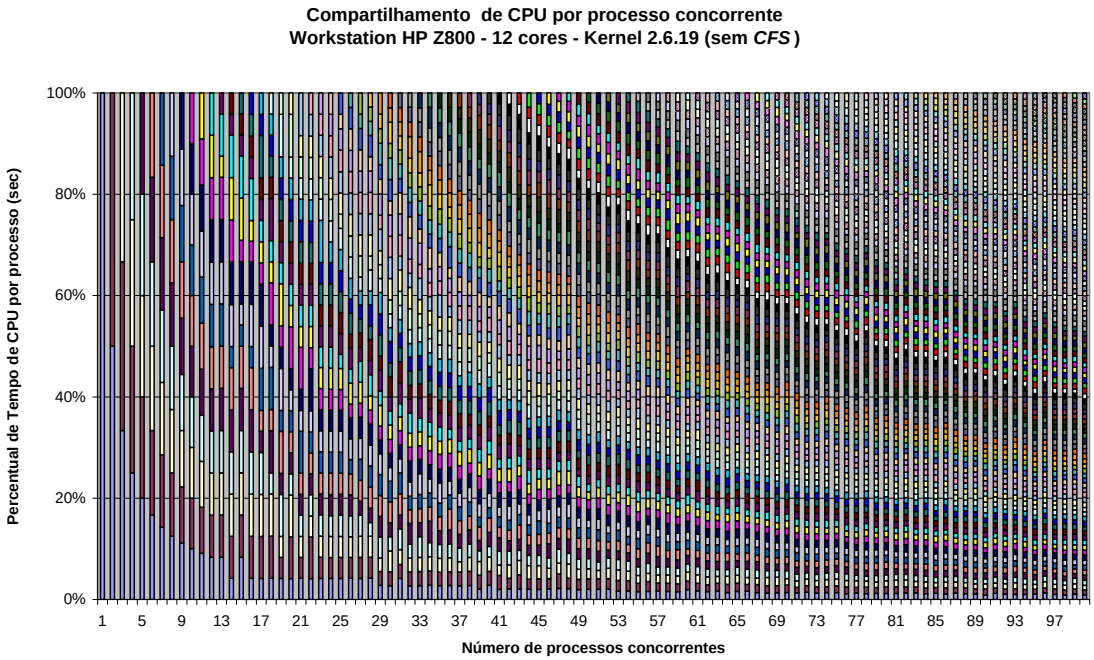

Figura 4-3 Gráfico do compartilhamento de CPU por até 100 processos concorrentes

100 processos a cada 60 segundos **com** o ***CFS***

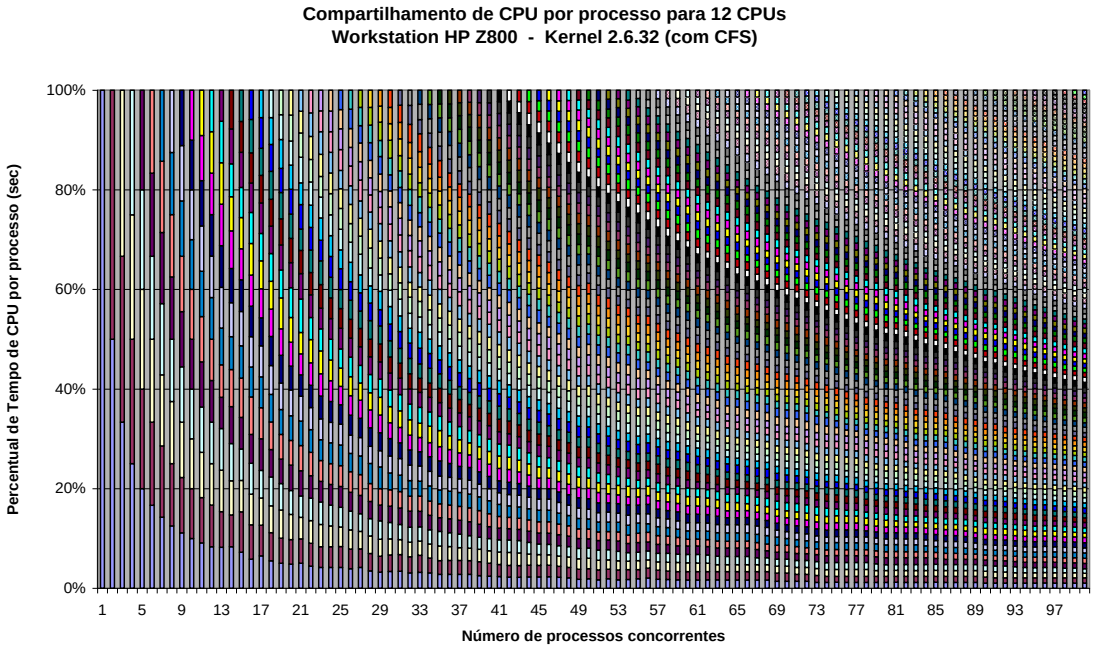

Figura 4-4 Gráfico do compartilhamento de CPU por até 100 processos concorrentes

Para as Figuras 4-3 e 4-4 valem os mesmos argumentos qualitativos discutidos para as Figuras 4-1 e 4-2. As variâncias para processos concorrentes de 13 a 21 (eixo dos X) com 100 processos de 60 segundos cada, são apresentadas abaixo.

Da Figura 4-3 (**sem *CFS***)

13	14	15	16	17	18	19	20	21
231,4533	238,1065	230,8552	211,8380	185,2650	152,9116	117,8118	77,24162	40,0333

Da Figura 4-4 (**com *CFS***)

13	14	15	16	17	18	19	20	21
38,1580	15,2073	21,2682	10,7625	7,2285	0,9813	6,9146	3,4267	16,5690

Para a variância populacional, encontramos os valores:

sem *CFS*, 95,5428

com *CFS*, 87,2787

Constatamos quantitativamente que, para as amostras de 13 a 21 processos concorrentes, a variância amostral é maior para o compartilhamento de tempo dos processos **sem *CFS*** de maneira significativa. Isto significa que nossas hipóteses sobre o comportamento qualitativo é justificável. Logo, podemos concluir, observando as variâncias das amostras para as quantidades de processos concorrentes de 13 a 21, bem como as variâncias populacionais (todas as amostras), que o escalonador ***CFS*** é menos injusto do que o escalonador anterior do Linux, ***O(1)***, porque as variações dos valores médios de compartilhamento de CPU medidas são significativamente maiores para o escalonador ***O(1)***, ou seja, os valores dos tempos de compartilhamento com ***CFS*** se aproximam dos valores médios.

Os resultados qualitativos e quantitativos dos gráficos das Figuras de 4-1 a 4-4, são significativos para nos incentivar a estudar mais profundamente o comportamento do escalonador ***CFS***. O escalonador ***CFS*** exibe um comportamento gráfico mais equânime para todos os processos a ele submetidos do que seu antecessor, ***O(1)***.

Verifica-se visualmente, pela análise dos gráficos das Figuras 4-1 a 4-4, que P_1 toma todo o tempo da primeira amostragem quando está sozinho; depois $P_1 + P_2$ juntos tomam a metade cada um do tempo da segunda amostragem, $P_1 + P_2 + P_3$ a terça parte cada um do tempo da terceira amostragem até, possivelmente *1/n* para cada *i-ésimo* processo P_i (onde i=1,2,...,n processos concorrentes), como se verificará.

Os gráficos das Figuras 4-1 e 4-2, respectivamente, revelam o comportamento dos escalonadores ***O(1)*** e ***CFS***. Os dois exibem visualmente algumas irregularidades na distribuição do consumo de CPU pelos processos à medida que o número de processos aumenta. Mas, no escalonador ***O(1)*** elas aparentam ser mais visíveis e destacáveis no comportamento da distribuição das cores, que parecem ficar mais embaralhadas visualmente. Estas irregularidades mostradas visualmente se intensificam com o número de processos concorrentes ligeiramente maiores que múltiplos do número de *cores* ou CPUs lógicas. Está forma é uma maneira ingênua de se tentar demonstrar um complexo comportamento do escalonador de maneira simples e de fácil compreensão. Baseia-se no argumento de que se a proporção de CPU de cada processo fosse o mais equânime possível a divisão seria o mais regular possível para cada instante de monitoração. Então, a fração de CPU de cada novo processo em um novo intervalo de tempo, sofreria uma queda cada vez mais e mais suave, cujo comportamento poderia ser visualizado no gráfico em questão.

No caso da presença do novo escalonador ***CFS*** com o *kernel* 2.6.32, verifica-se a maior harmonia na distribuição média das cores que representam proporcionalmente o tempo médio de CPU consumido por cada um dos processos concorrentes. Ainda assim, verifica-se também a preservação de algumas irregularidades mais sutis que as provocadas pelo escalonador anterior, **sem** o ***CFS***. Estas irregularidades relativamente suaves devem-se a necessidade de maior gasto de gerenciamento do próprio escalonador ***CFS*** e das ingerências mínimas mas cada vez mais significativas, referentes aos custos da monitoração para análise da equidade do ***CFS*** e do balanceamento de carga de tempos em tempos, por exemplo, definidas pelo intervalo de monitoração e critérios internos de parametrização do próprio balanceador do escalonador ***CFS***. Na realidade, há mais processos concorrentes do que os **LI**s do experimento 4.3. Outros processos, incluindo os de monitoração e do próprio sistema, concorrem pelos mesmos recursos de CPU, induzindo a estas "desproporcionalidades" visualmente identificáveis. Mas, mesmo assim, o ***CFS*** se mostra mais regular do que o escalonador anterior. Uma maior e melhor suavização hipotética do comportamento do

particionamento percentual dos tempos dos processos representados pelas suas respectivas cores, levaria mais precisamente a um comportamento funcionalmente proporcional a *1/n*. Isto mostraria que este novo escalonador, quando comparado aos seus antecessores, é mais equânime e se comporta mais próximo de um escalonador ideal.

Admite-se que a heurística simples e a representação da estrutura interna em árvores rubro-negras, que são autobalanceáveis e individualizadas por CPU, demonstram uma maior escalabilidade e menores custos de escalonamento bem como um maior aproveitamento do tempo de CPU para ou processos. Portanto, o novo escalonador ***CFS*** disponibiliza recursos computacionais de CPU com custos mais baixos de escalonamento.

A seguir, baseado nos dados experimentais do experimento anterior, com 100 processos, verifica-se a partir da análise quantitativa da variância dos tempos médios de CPU de cada processo, o comportamento da distribuição quantitativa do tempo médio consumido por cada processo concorrente em relação ao tempo total de todos os processos concorrentes até aquele momento, em cada intervalo de tempo de 60 segundos, no qual são feitas as medições. Serão analisadas as informações obtidas para cada configuração do *kernel* **com** e **sem** o escalonador ***CFS***.

A Figura 4-5, correspondente ao experimento **sem** o ***CFS***, mostra claramente irregularidade entre os dados experimentais e uma curva teórica traçada a partir de uma suavização via *spline cúbico*[8]. A curva experimental é nitidamente irregular e não suave tanto no início, quando da carga de todas as CPUs, quanto em outros tantos pontos do experimento. Destaquem-se os de 38 e 66 processos concorrentes, além do intervalo de 30 a 80 processos concorrentes.

A Figura 4-6, correspondente ao experimento **com** o ***CFS***, que é graficamente mais suave que aquela obtida **sem** o ***CFS*** da Figura 4-5 anterior. O resultado experimental **com *CFS***, exibido na Figura 4-6, coincide com a suavização teórica via *spline cúbico* em quase todo o seu intervalo de amostragem. Observe-se que a média também coincide com as cargas iniciais até que todas as CPUs estejam ocupadas (12 *cores*), diferentemente do caso anterior, mostrado na Figura 4-5.

[8] Spline cúbico é um método matemático para ajuste da melhor ou dita mais suave curva que passa por *n* pontos através de um polinômio de terceiro grau (cúbico).

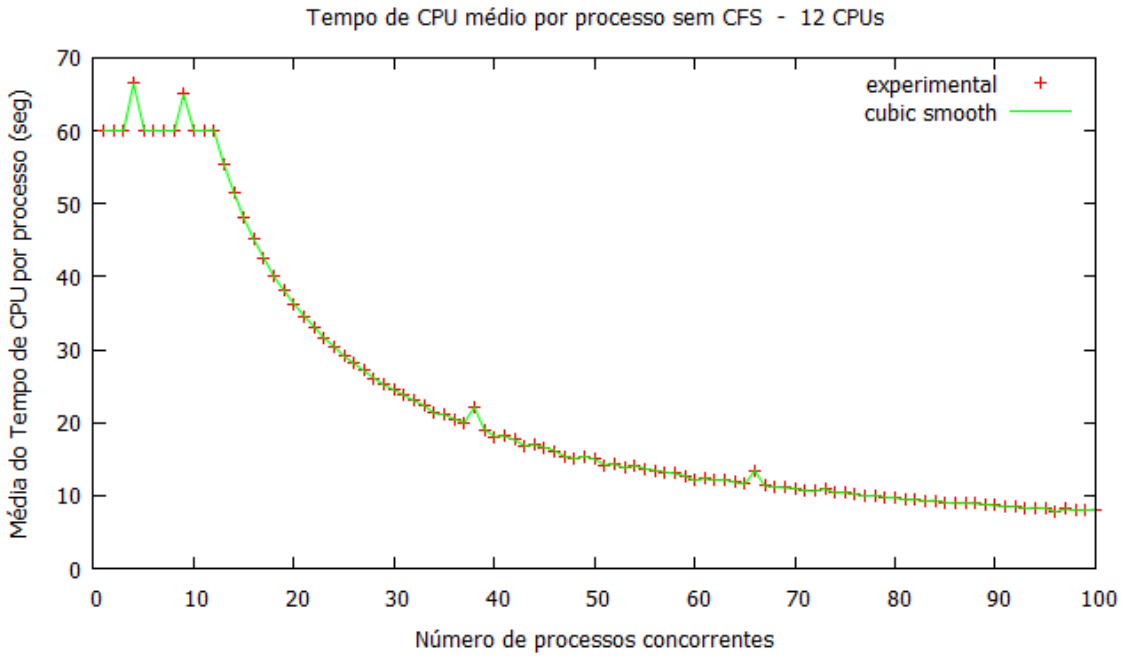

Figura 4-5 Gráfico do comportamento médio de consumo de CPU por processo em cada instante de tempo.

O gráfico da Figura 4-6 exibe o comportamento quantitativo da distribuição média de tempo de CPU por processo com o novo escalonador ***CFS*** com *kernel* 2.6.32 para um equipamento com 12 CPUs, onde a curva "cubic smooth" foi obtida através do método spline cúbico:

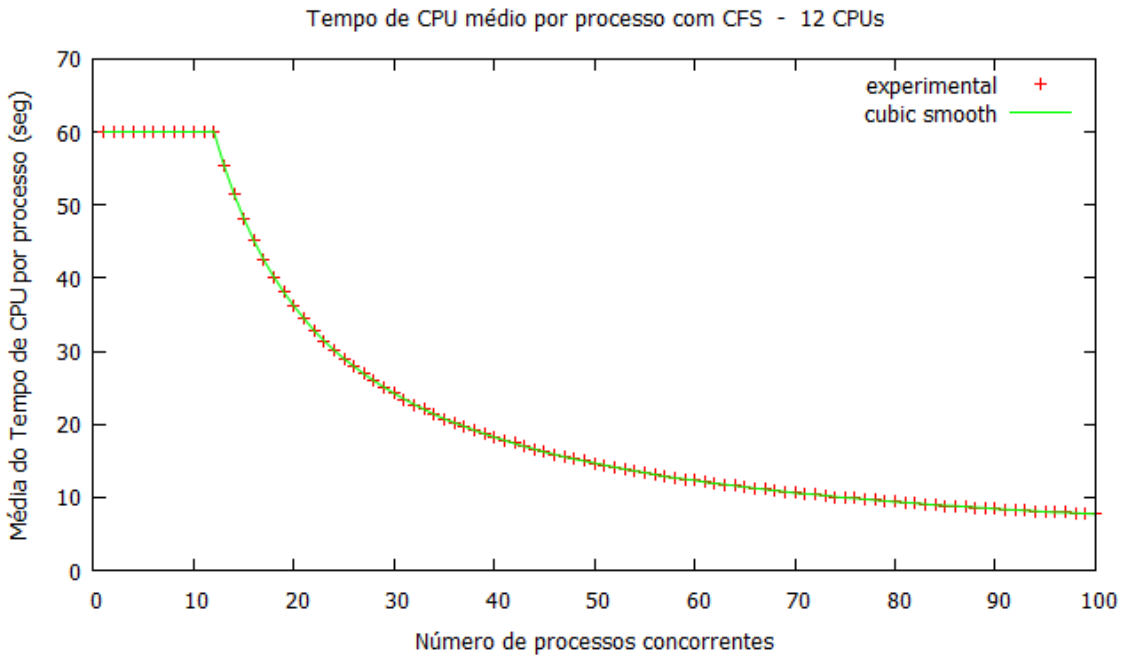

Figura 4-6 Gráfico do comportamento médio de consumo de CPU por processo em cada instante de tempo

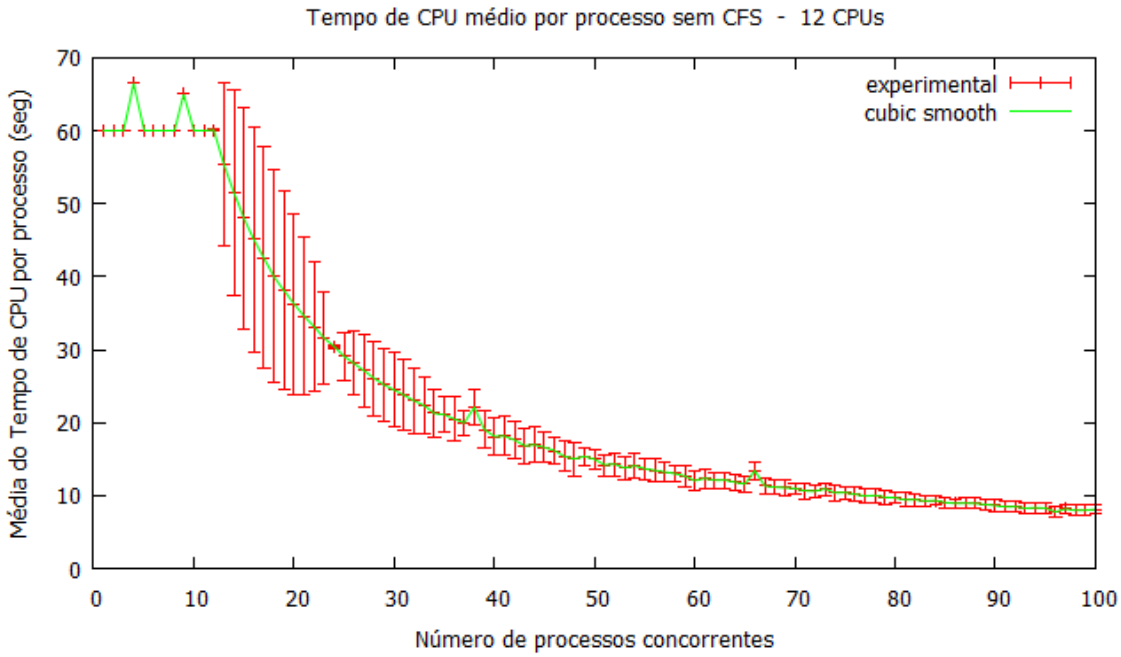

Figura 4-7 Compartilhamento de CPU por processo com barra de erro

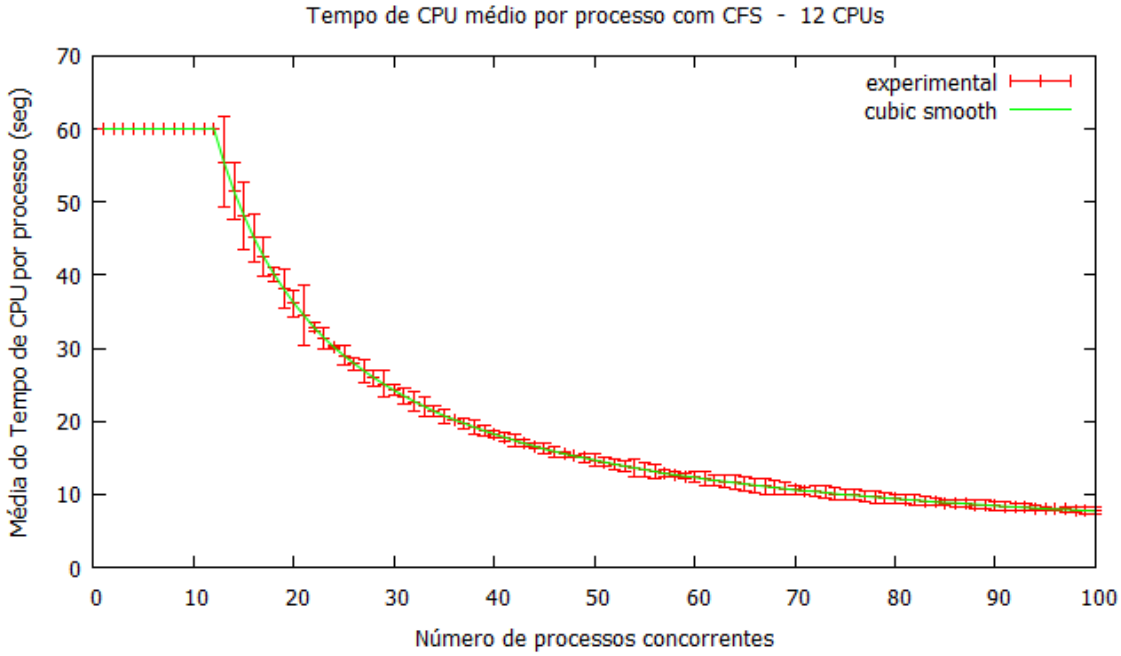

Figura 4-8 Compartilhamento de CPU por processo com barra de erro

As Figuras 4-7 e 4-8 representam as distribuições das médias de cada tempo de CPU por processo concorrente com suas barras de erros dadas pelas variâncias

associadas ao desvio padrão das medidas em cada intervalo de tempo. Note-se que estes erros são mais acentuados no escalonador **sem *CFS***, como pode ser visto na Figura 4-7. Isto representa que os tempos médios de consumo de CPU compartilhados pelos processos concorrentes no ***CFS***, em cada intervalo de tempo, são mais próximos uns dos outros do que no *kernel* **sem *CFS***. Isto significa, quantitativamente que o *kernel* **com *CFS*** é menos injusto que o **sem *CFS***. Em outras palavras, a análise quantitativa através das variâncias corrobora as hipóteses qualitativas, através da análise gráfica.

4.5 AVALIANDO A EQUIDADE DO *CFS* EM LARGA ESCALA – VARIANDO O NÚMERO DE *CORES*

Este experimento objetiva verificar o comportamento da distribuição de tempo de processador dentre processos concorrentes do tipo **LI**. O tempo medido é fornecido em um arquivo de saída via o comando *time* do Linux, mostrando o tempo de relógio de parede (*wallclock time*), o tempo efetivo consumido pelo usuário (*user time*) e, finalmente, o tempo consumido pelo sistema para cada processo per si (*sys time*), em um determinado intervalo de tempo. Os dados são condensados e tabulados, obtendo-se gráficos qualitativos e quantitativos representativos dos fenômenos.

Os procedimentos deste experimento foram realizados em uma *Workstation* HP Z800 com dois processadores físicos (*Dual Processor*) Intel *Nehalem Hexacore* (**6** CPUs). Os processadores são ativados/inibidos diretamente colocando-se 0 (ligar) ou 1 (desligar) em seus respectivos arquivos ***online*** residentes no diretório */sys/devices/system/cpu/cpu***X** (onde, **X** = 0, *1*, *2*, ... , *n -1* cpus do sistema). Pode-se definir também intervalos contíguos de processadores físicos no arquivo ***online*** do diretório */sys/devices/system/cpu*.

Foram realizados testes para 1 (cpu 0), 2 (cpus 0-1), 4 (cpus 0-3), 6 (cpus 0-5) e 12 (cpus 0-11) com *kernel* **sem *CFS*** e **com *CFS***. Os resultados são exibidos nos seguintes gráficos aos pares para cada *kernel* **com** e **sem** o ***CFS***.

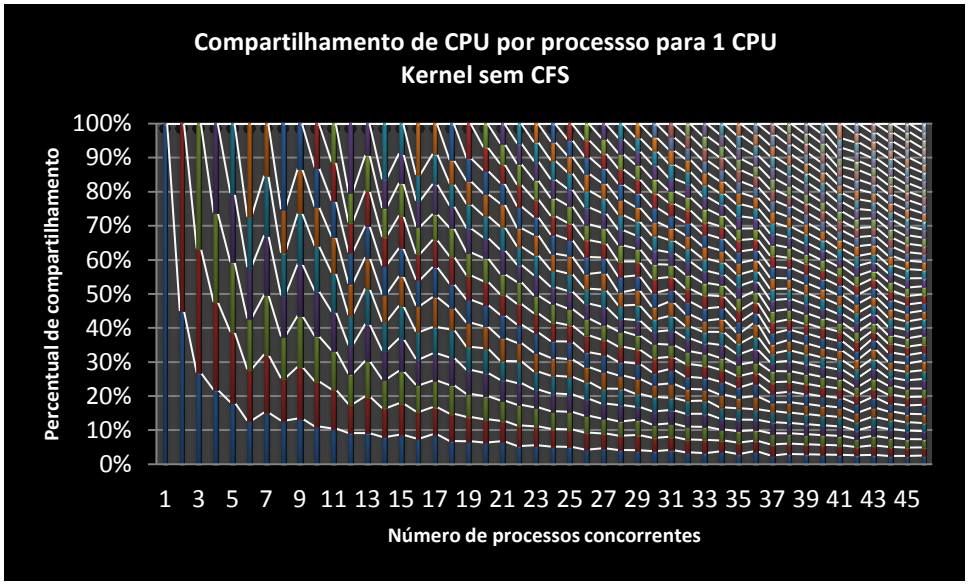

Figura 4-9 Distribuição de % CPU entre 46 processos sem CFS para 1 CPU.

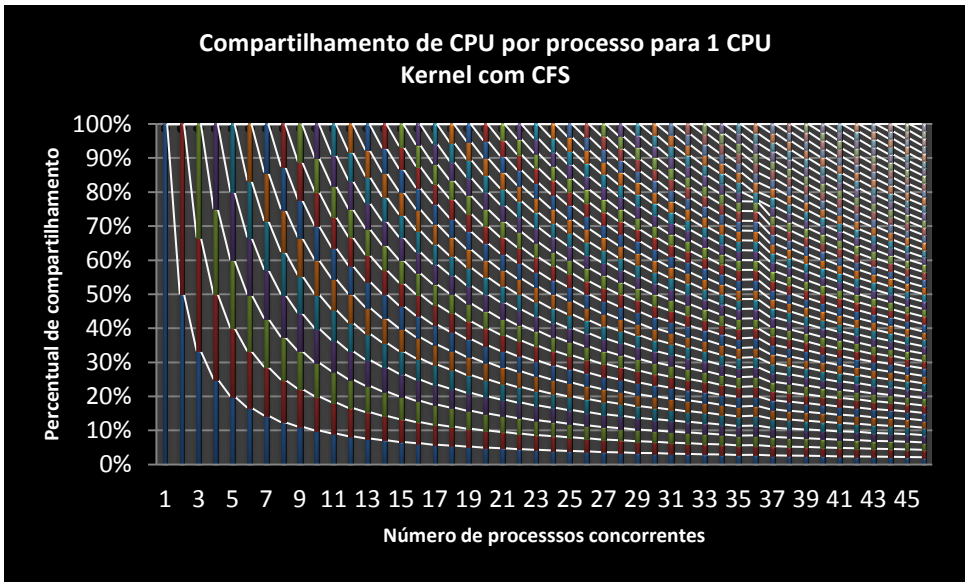

Figura 4-10 Distribuição de % CPU entre 46 processos com CFS para 1 CPU.

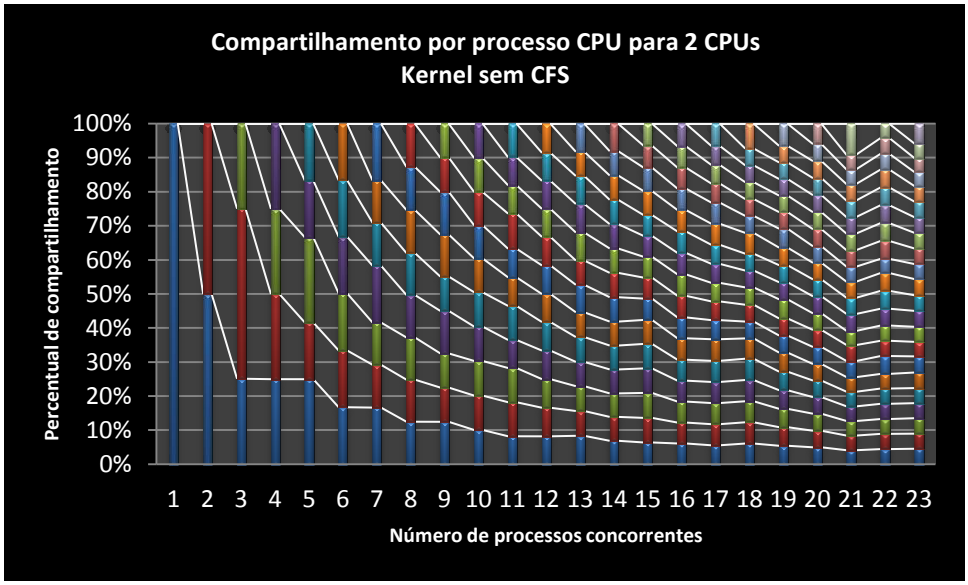

Figura 4-11 Distribuição de % CPU entre 23 processos sem CFS para 2 CPUs.

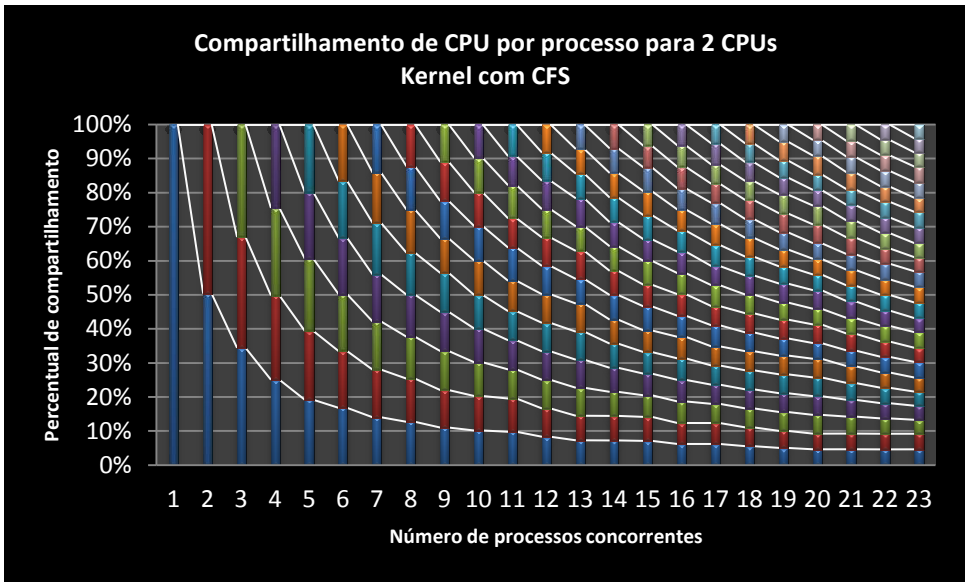

Figura 4-12 Distribuição de % CPU entre 23 processos com CFS para 2 CPUs.

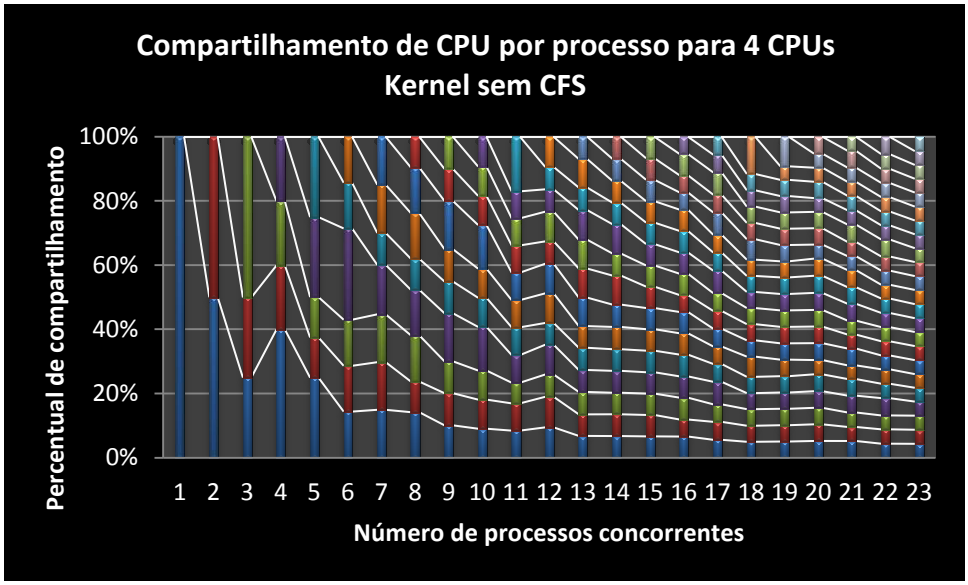

Figura 4-13 Distribuição de % CPU entre 23 processos sem CFS para 4 CPUs.

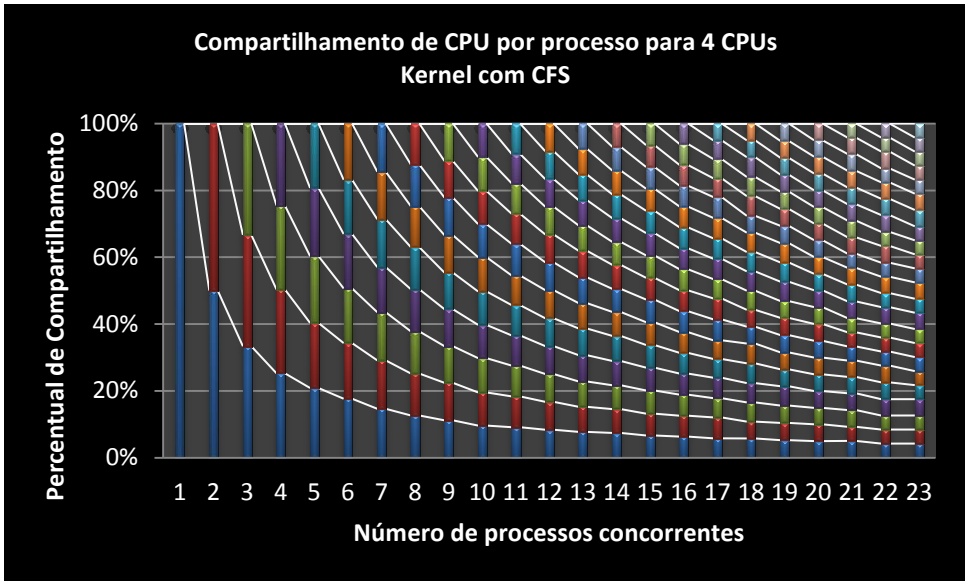

Figura 4-14 Distribuição de % CPU entre 23 processos com CFS para 4 CPUs.

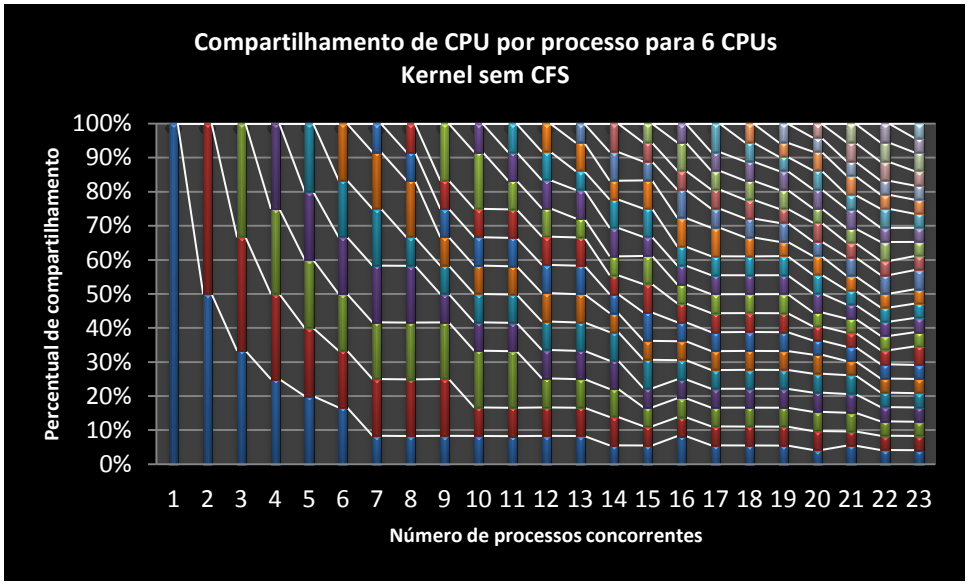

Figura 4-15 Distribuição de % CPU entre 23 processos com CFS para 6 CPUs.

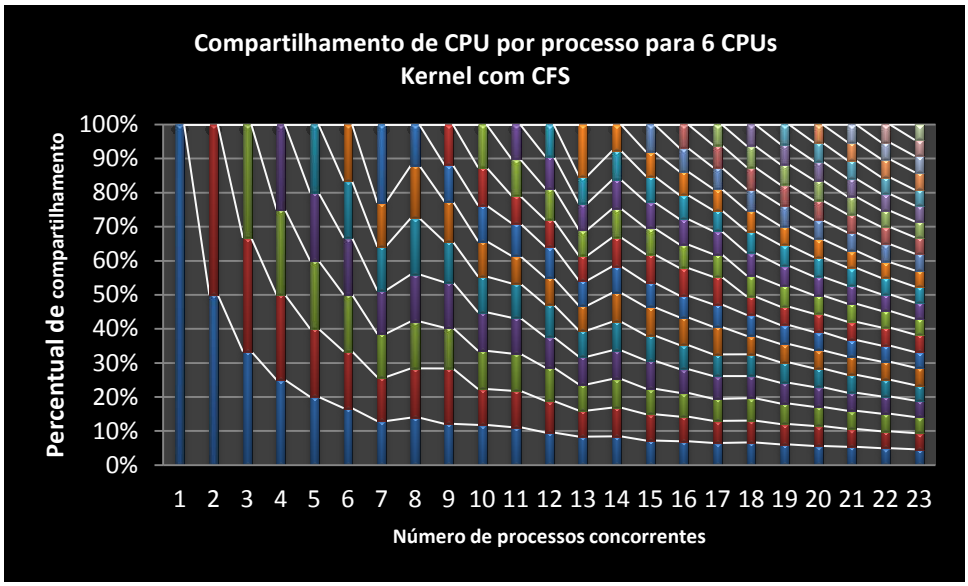

Figura 4-16 Distribuição de % CPU entre 23 processos com CFS para 6 CPUs.

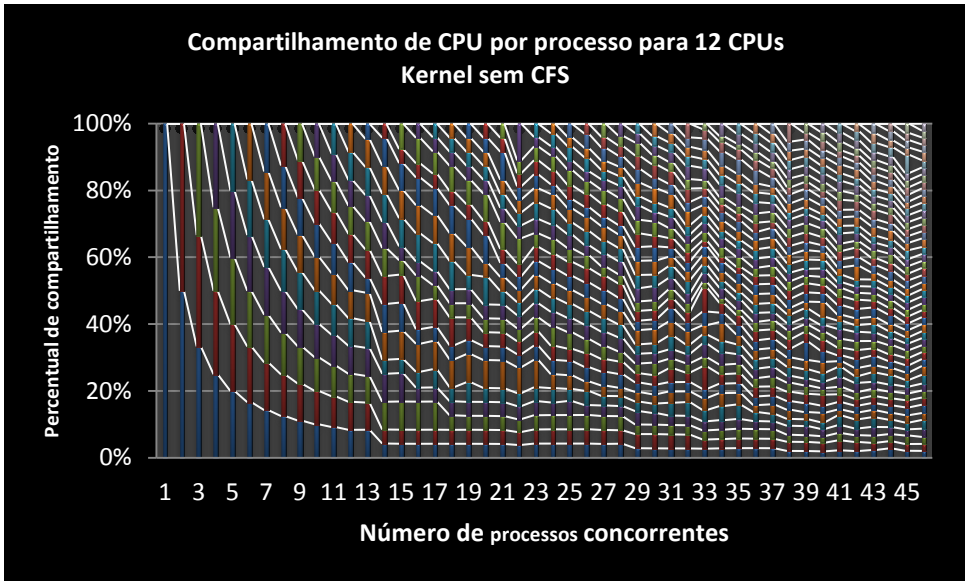

Figura 4-17 Distribuição de % CPU entre 46 processos sem CFS para 12 CPUs.

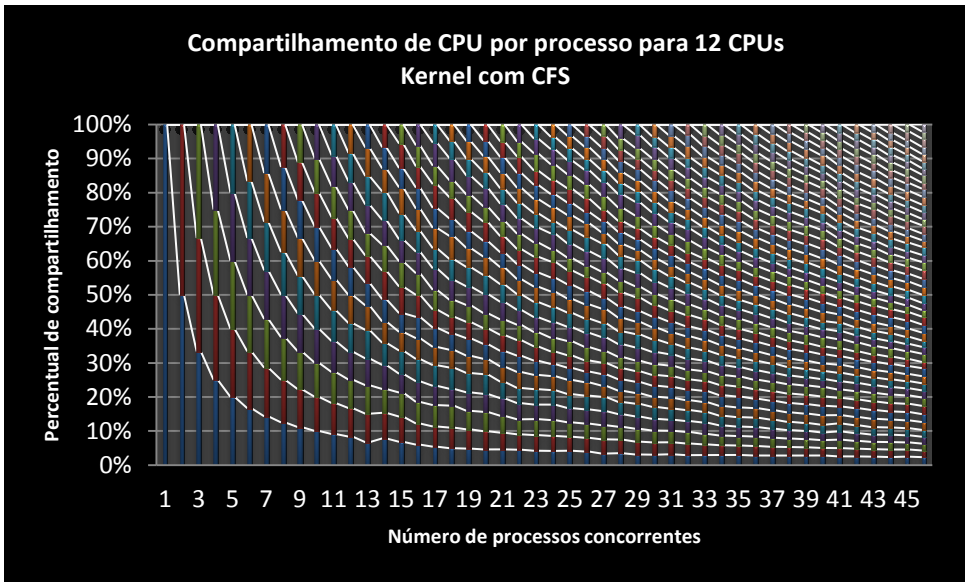

Figura 4-18 Distribuição de % CPU entre 46 processos com CFS para 12 CPUs.

CPU	CFS	Número de processos concorrentes										
1		2	3	4	5	6	7	8	9	10	11	12
var	com	0,00020	0,00013	0,00022	0,00033	0,000297	0,00050	0,00012	0,00094	0,00024	0,00014	0,00030
	sem	1,69280	1,60623	0,23826	0,06180	2,584706	0,06166	1,68952	0,05549	0,04851	0,00944	1,031173
2		2	3	4	5	6	7	8	9	10	11	12
var	com	0,06125	0,31863	0,08795	0,21156	0,00196	0,07934	0,01294	0,06905	0,00611	0,10906	0,00075
	sem	0,00020	75,05723	0,03746	7,61116	0,00250	1,68847	0,00318	0,66017	0,00045	0,26436	0,00280
4		2	3	4	5	6	7	8	9	10	11	12
var	com	0,00005	0,00003	0,14029	0,57997	0,385628	0,18788	0,03292	0,02467	0,05748	0,04326	0,002154
	sem	0,00003	0,00003	0,00012	0,00010	80,46969	6,62242	9,62069	2,00190	4,22816	1,41631	19,90905
6		2	3	4	5	6	7	8	9	10	11	12
var	com	0,00005	0,00003	0,00002	0,00002	0,00003	0,00003	0,00003	0,00002	0,00011	0,00019	0,03922
	sem	0	0,00003	0,00004	0,00005	0,00003	54,28123	66,09996	58,716020	40,95298	21.013	0,01539
12		13	14	15	16	17	18	19	20	21	22	23
var	com	7,52046	6,48284	10,04212	0,34777	3,10458	6,19638	1,42922	3,49283	0,79484	0,38116	0,67835
	sem	36,85643	49,68727	56,92331	55,41369	52,35331	51,60823	38,11009	38,01497	29,11971	51,85290	11,21220

Tabela 4-4 Análise da variância em função do número de CPUs com e sem CFS

A Tabela 4-4 mostra um resumo dos resultados, apresentando o cálculo de algumas variâncias representativas do comportamento da distribuição do tempo de CPU quando a quantidade de CPU varia de 1, 2, 4, 6 e 12 **com** e **sem** o escalonador ***CFS***. Estes resultados correspondem aos dados mostrados nas Figuras de 4-9 a 4-18, onde podem ser vistas as suavidades das linhas dos gráficos obtidos **com** o ***CFS***, diferentemente dos obtidos **sem** o ***CFS***. Está análise do padrão de comportamento do ***CFS*** em grande escala (muitas CPUs e processos) pode ser interpretado quantitativamente com os resultados da Tabela 4-4, onde são mostradas as variâncias nos seus pontos mais críticos por simplicidade. A regularidade da partilha de CPU pelos processos no caso do experimento **com *CFS*** é visível. No outro caso, **sem *CFS***, elas são graficamente acentuadas, pelo acompanhamento gráfico ou, quantitativamente quando são consideradas as variâncias em cada caso, exibida na Tabela 4-4.

Pode-se concluir deste experimento, que o escalonador ***CFS*** é mais justo que o seu antecessor tanto do ponto de vista qualitativo, mostrado pelo seu comportamento através dos gráficos das Figuras 4-9 a 4-18, como do quantitativo, conforme os resultados das variâncias calculadas referentes aos dados das médias de tempos de CPU destinadas aos processos.

4.6 OVERHEAD DO SISTEMA

4.6.1 Caso 1 – processos LI

Este experimento foi concebido com o objetivo de se verificar o *overhead* do sistema, isto é, o tempo necessário para conclusão de tarefas paralelas do tipo **LI** com uma intervalo de espera de 100 segundos após o qual o *loop* dispara a criação dos processos em *background*. Os processos são submetidas em um *loop* inicial e o sistema aguarda um determinado tempo de 100 segundos para o processamento da criação e execução dos processos de *loop infinito*, findo o qual é efetuado o computo do tempo total de processamento para criação de todos os processos. São executados *jobs* para 12, 24, 36, 48, 72, 96, 192, 384, 768 e 1000 processos concorrentes. Os resultados estão representados nos seguintes gráficos das Figuras 4-19 e 4-20, onde o tempo de conclusão das criações dos processos de cada *job* é plotado em função do número de processos do *job*:

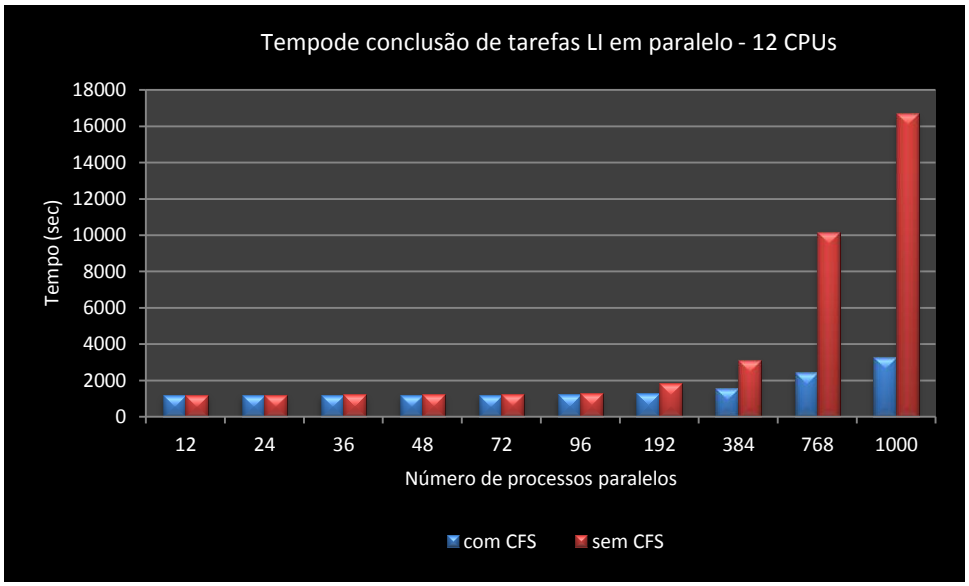

Figura 4-19 Gráfico que representa o consumo total de CPU para atingir uma mesma quantidade de tarefas em um intervalo de tempo de relógio pré-definido

Pela análise do gráfico da Figura 4-19, podemos inferir que os escalonadores **sem *CFS*** e **com *CFS***, podem ser interpretados como aproximadamente equivalentes para poucas dezenas de processos, mas à medida que o número de processos aumenta

para a proporção de poucas centenas destes, o ***CFS*** é mais eficiente e capaz de tratá-los de modo mais igualitário, minimizando o tempo necessário para uma tarefa de muitos processos paralelos em menor tempo total. No escalonador anterior ao ***CFS***, o custo de gerenciamento é muito prejudicial bem como a iníqua distribuição de CPU entre os processos concorrentes. Isto implica que aqueles processos de criação concorrem de modo desigual com os processos já criados além dos de monitoração, retardando a conclusão dos processos como um todo. Aliado a isto, quando todos os processos de cada job forem criados e só eles existirem em execução além dos de monitoração, estes últimos, que também são disparados em sequência, terão que concorrer e aguardar por todos aqueles outros em execução, que não são eliminados por simplicidade (mas poderíamos fazê-lo). Como podemos interpretar pelo gráfico da Figura 4-19, o tempo total é muito maior no escalonador anterior, inclusive para a criação e determinação dos identificadores dos processos em paralelo com veremos mais a frente.

4.6.2 Caso 2 – processos CF

Este experimento visa verificar o *overhead* do sistema a devido a escolha do escalonador. São submetidos processos com carga definida de aproximadamente 100 segundos de CPU pelo método **CF**, submetidos em um *loop* de criação inicial. Após o término da aplicação, isto é, de todos os N=12, 24, 36, 48, 72, 96, 192, 384, 768 e 1000, processos que fazem parte da aplicação, é computado o tempo total de CPU do processamento. Os testes são efetuados para o escalonador **com** e **sem** o ***CFS***.

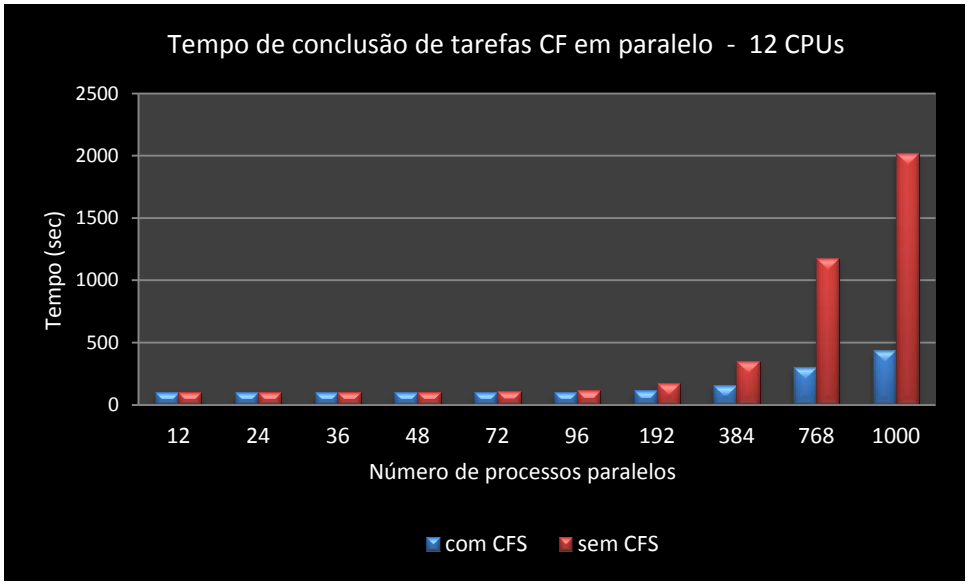

Figura 4-20 Gráfico que representa o tempo de relógio para conclusão de uma mesma quantidade de tarefas em um intervalo de tempo pré-definido (aplicação CF)

Fundamentalmente, valem os argumentos da Seção 4.6.1. Logicamente observa-se que este tempo real é bem menor que o tempo de consumo de CPU total, pois este leva em conta todas as CPUs do sistema, no caso *multicore* de 12 núcleos em duas CPUs físicas, mas demonstra o excessivo consumo devido às características de gerenciamento do escalonador. Claramente, é mostrada a vantagem do ***CFS*** para mais de cem processos concorrentes em relação ao escalonador anterior. A questão fundamental é que o ***CFS*** tende internamente, por concepção e construção, a privilegiar os processos mais novos, pois estes sempre terão os menores *vruntime* de suas filas de execução, forçando-os a receber maior quantidade de recursos, CPU no caso, até que quase todos fiquem com proporções parecidas de CPU enquanto o escalonador anterior tende a favorecer os mais velhos como um todo. De qualquer forma, o ***CFS*** é mais eficiente e equânime, como se propõe.

Pela análise dos gráficos das Figuras 4-19 e 4-20, observa-se que as aplicações **LI** apresentam *overhead* com valores muito maiores que as aplicações **CF**. Isto se explica simplesmente porque as aplicações **CF** são finitas e não cumulativas após a execução da carga fixada de tempo de CPU para cada uma delas (100 segundos). Por outro lado, as **LI** tendem a concorrer indefinidamente, pois não são eliminadas exceto no final, quando o teste elimina todas as aplicações após a criação da última e um pequeno tempo de CPU extra de execução de 100 segundos. Os resultados obtidos

mostram que o CFS escala muito melhor para um grande número de processos que o seu antecessor, com um *overhead* bem menor.

4.7 MUDANÇA DA LATÊNCIA MÍNIMA DO ESCALONADOR *CFS*

4.7.1 Avaliação do tempo de execução

A latência mínima do escalonador, dada pelo parâmetro *kernel.sched_latency_ns* em unidades de tempo de nanossegundos, é definida em tempo de *boot* ou dinamicamente reconfigurável via comando ***sysctl*** . A latência mínima do escalonador é o tempo mínimo garantido pelo escalonador para a execução dos processos que não exigem interrupções provocadas por necessidades de Entrada e/ou Saída ou outras que se fizerem necessárias ao processo. A menos que estas aconteçam, o escalonador tende a garantir o tempo definido na latência mínima para o processo atualmente em execução. Portanto, pode-se observar a consequência na definição da latência mínima como sendo a de seu valor máximo, ou seja, um bilhão de nanossegundos, isto é, um tempo de 1(um) segundo. Os outros parâmetros, *kernel.sched_min_granularity_ns e kernel.sched_wakeup_granularity_ns* também foram definidos com o mesmo valor. Cabe lembrar que o intervalo de variação destes parâmetros não está documentado. A análise do impacto do ajuste do parâmetro ***sysctl*** do escalonador pode ser visto em [84].

O resultado pode ser observado no gráfico da Figura 4-21.

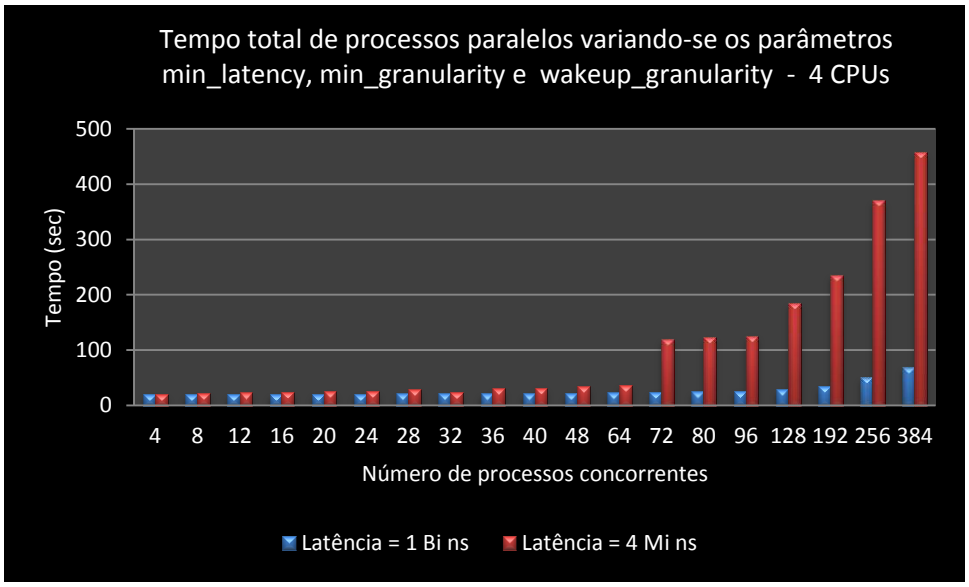

Figura 4-21 Tempo total de processos paralelos (CF) por número de processos para um latência mínima com seu máximo valor, 1 segundo

O tempo total de processamento de um dado número de processos concorrentes tende a aumentar com um número de processos necessariamente, pois os processos que obtiverem o controle da CPU também tenderam a preservá-lo até o seu limite de 1 segundo. Este limite superior é dado pelo parâmetro *kernel.sched_latency_ns*, definido dinamicamente ou pelo *default* do escalonador ***CFS*** para a plataforma, em tempo de boot. Como consequência, isto implicará em uma ruptura com o objetivo de maior interatividade e sensação de tempo de resposta do usuário, pois, os processos que detiverem o controle de alguma CPU, sob o controle e consentimento do escalonador ***CFS***, tenderá a mantê-la, impedindo ou retardando, assim que outro ou um novo processo possa fazê-lo. Ou seja, todos os processos deverão consumir muita CPU. Portanto, para que um número de tarefas seja concluído deve-se consumir muito processador até que todos terminem, incluindo-se os *jobs* de criação e monitoração, já que, apesar do baixo consumo relativo, também concorrem com os outros processos por CPU. Como consequência, os processos interativos serão penalizados com atendimento não imediato quando concorrerem com muitos processos CPU-*bound*, por exemplo.

4.7.2 Avaliação do número de trocas de contexto em função do tamanho do executável

A troca de contexto é o método ou procedimento através do qual o sistema operacional, via escalonador, substitui um processo ou tarefa em execução por outro. Isto é implementado salvando-se o estado do programa em execução através do conteúdo de seus registradores e recuperando-se o estado anterior do programa a executar através do recarregamento de seus conteúdos nos registradores do sistema. Por último, é carregada a parte executável do processo a executar em memória e o controle da CPU é passado para a primeira instrução do executável carregado. Este procedimento tem um custo de tempo para o escalonador e pode ser prejudicial para a tarefa em execução se esta sofrer uma elevada troca de contexto e para as outras tarefas como um todo. Minimizar a troca de contexto pode ser considerada um tarefa relevante para o desempenho do sistema no geral, pois evita-se assim as desnecessárias trocas e substituições de estados e conteúdos de registradores de um código em execução para outro código a executar. Procura-se garantir assim, que o executável esteja o mais próximo da CPU no tempo. Em outras palavras, busca-se tirar proveito do princípio da localidade.

No gráfico da Figura 4-22 é verificada a vantagem da minimização da troca de contexto quando alteramos o parâmetro de controle da latência mínima do escalonador ***CFS*** de 20 milhões de nanossegundos (parâmetro *default)* para 1 bilhão de nanossegundos. Isto é benéfico para todos os processos em geral mas, de forma mais significativa para processos com grandes módulos de carga. Para pequenos módulos de carga executáveis, o ganho também é importante, mas estes tem maior probabilidade de já estarem na memória cache do sistema (em um cache de 12MB podem ser armazenados poucas centenas de pequenos módulos executáveis de 4K, por exemplo). Para executáveis grandes, outro benefício seria a desnecessária movimentação de megabytes de código, se preciso, a cada troca de contexto, ou seja, mais recursos de CPU são disponibilizados para os processos e não para outras operações do sistema.

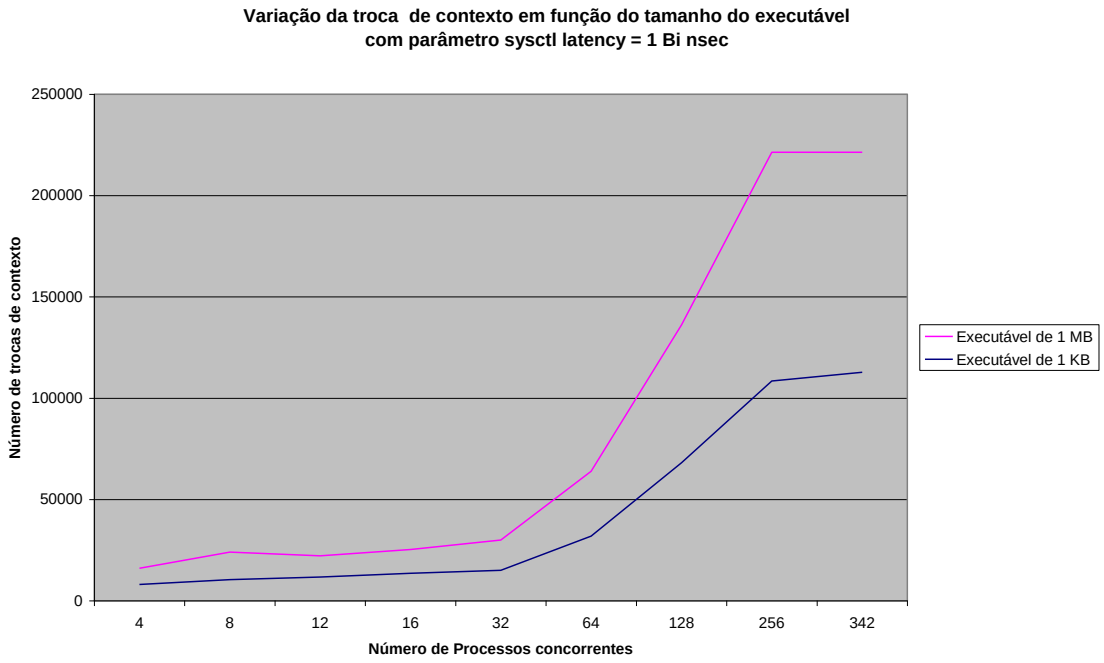

Figura 4-22 Gráfico da variação do número de trocas de contexto em função da latência mínima definida com o seu máximo valor (1 segundo).

Explicitamente, pela análise do gráfico, pode-se inferir que o número de trocas de contexto é maior para executáveis com tamanhos maiores, pois estes tem menor probabilidade de ficarem alocados na memória cache dos processadores, mesmo que estes tenham 12 MB de cache (Intel Xeon Nehalem), diferentemente dos de menor tamanho (4KB). Caches maiores podem conter mais processos que correspondam a módulos de carga menores sem a necessidade de substituí-los por outros imediatamente

até que o memória cache fique cheia e os algoritmos de atualização da cache tenham que atuar provocando obrigatórias trocas de contexto.

A contabilização do número de trocas de contexto foi efetuada através do utilitário *sched_stat* executado em determinados intervalos de amostragem de 1 segundo.

4.7.3 Avaliação da troca de contexto com e sem *CFS*

A Figura 4-23 mostra o comportamento do número de trocas de contexto para os dois escalonadores, da versão do *kernel* 2.6.18 e da versão 2.6.32, da qual faz parte o ***CFS***. Esta última com parâmetros *default* ou com sua latência máxima definida para 1 segundo.

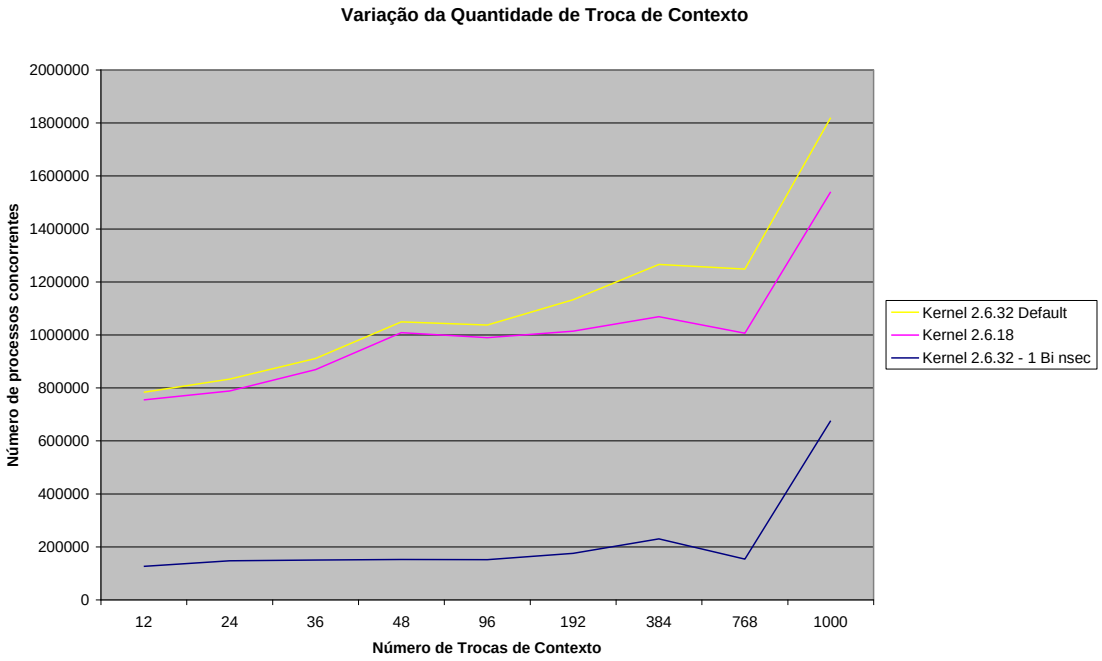

Figura 4-23 Gráfico de variação do número de trocas de contexto de processos paralelos em função do escalonador com opção default e latência mínima máxima do *CFS*

Claramente verifica-se que o comportamento do número de trocas de contexto do *kernel* **sem *CFS*** do **com *CFS***, são muito próximos com um menor número para o correspondente ao escalonador mais antigo porque a granularidade da latência mínima do ***CFS***, por *default*, é muito menor para aumentar a interatividade do sistema, ou seja, o ***CFS*** permite que os processos possam ser chaveados mais rapidamente, provocando

assim uma mudança de contexto mais frequente que no escalonador anterior, **sem *CFS***. Porém, diferentemente do escalonador mais antigo, o ***CFS*** também permite que se possa fazer dinamicamente uma minimização do número de trocas de contexto através da alteração do parâmetro *kernel.sched_latency_ns* do escalonador, que pode ser alterado para um valor de até 1 segundo (dado em unidades de nanosegundos). Esta alteração é muito significativa e, diminui o número de trocas de contexto para os processos em execução ou a executar no sistema. Cabe ressaltar que reduzir a troca de contexto implica em liberar maior tempo de CPU para todos os processos ou aplicações. Por outro lado, aumentar a latência mínima do escalonador implica em diminuir a interatividade do sistema ou a sensação de tempo de resposta experimentada pelo usuário. Portanto, deve ser avaliado o tipo das aplicações que serão executadas e então, proceder com a escolha mais adequada de parâmetros, que pode ser feita a qualquer instante, no escalonador ***CFS***. Com base neste experimento, conclui-se que é possível escolher o mais adequado para processos ou os e/ou aplicações específicas, como as CPU intensivas. Basta alterar dinamicamente os parâmetros reconfiguráveis do escalonador ***CFS***, a qualquer tempo. Esta abordagem é de enorme importância para ambientes do tipo HPC ou do tipo Nuvem, ou outros sistemas que necessitem de muita CPU. Sistemas de alto desempenho, em geral, são CPU intensivos e não podem se dar ao luxo de pequenos desperdícios de recursos devido à escala (milhares, dezenas de milhares de CPUs).

4.8 RESUMO

Os resultados dos experimentos deste Capítulo comprovam que, tanto do ponto de vista da pequena ou da grande escala, o ***CFS***, é mais justo que o escalonador anterior, ***O(1)***. Eles também revelam um excelente balanceamento espacial de carga, isto é, o ***CFS*** procura distribuir a carga do sistema uniformemente entre as CPUs, independentemente do número de processos ser menor ou maior do que o número de CPUs. Portanto, o ***CFS*** é bastante escalável e, logo, mais eficiente do que o seu antecessor. O comportamento qualitativo do ***CFS***, exibido através dos gráficos de compartilhamento percentual de CPU, ou o quantitativo, através da análise das variâncias das médias dos tempos de CPU dos processos, revela que o ***CFS*** distribui de modo mais igualitário os recursos do que o seu antecessor. O preço de sua justiça praticada pelo escalonador é o aumento do número de trocas de contexto, pois ele distribui muito mais vezes pequenas porções de CPU entre os processos. Neste

Capítulo, comprova-se que o ***CFS*** é dinamicamente reconfigurável e sua reconfiguração pode se útil para certas aplicações como HPC [85], por exemplo. Mostra-se que o ***CFS*** também se autorreconfigura para garantir seus objetivos de justiça, aumentando ou diminuindo parâmetros do escalonador para se adequar a adições ou subtrações de CPU. O *overhead* do ***CFS*** é bem menor do que seu antecessor, principalmente para altas cargas do sistema. Constata-se que aumento da latência e granularidade mínimas do escalonador ***CFS*** pode beneficiar as aplicações CPU intensivas, pois diminuem o número de trocas de contexto, por exemplo.

CAPÍTULO 5 – GRUPOS DE CONTROLE DO CFS

Nos capítulos anteriores foi estabelecida a motivação para o escalonador ***CFS***, suas estruturas internas e os pré-requisitos do *kernel* do sistema operacional do Linux: frações de tempo em nanossegundos, *kernel tickless* [86] e algoritmos de escalonamento proporcional. Portanto, podemos concluir até aqui que o escalonador ***CFS*** é tão equânime quanto se propõe. Em outras palavras, é suficientemente equânime com proporcionalidade unitária para todos os processos na média, guardadas as respectivas tolerâncias para o gerenciamento das filas e processos, e não apenas o consumo de CPU dos próprios processos. Porém, o custo de gerenciamento pode ser considerado mínimo, dada a heurística simples ou baseada na divisão aritmética do tempo de CPU proporcional entre os processos concorrentes.

Na verdade, esta proporcionalidade é de peso unitário não para cada processo, mas sim para o conjunto de processos concorrentes de cada identificador de usuário (***userid***) distinto.

Neste capítulo será apresentada a implementação de grupos de controle no Linux, que permitem definir agrupamentos e compartilhamento de recursos computacionais com pesos não unitários, ou seja, particionamento de recursos em conformidade com porções definidas pelo administrador do sistema para os usuários, aplicações, grupos de usuários e/ou subgrupos destes definidos em arquivos de configuração e dinamicamente reconfiguráveis.

5.1 CGROUPS - CONTROL GROUPS

Como será verificado nos experimentos a seguir, o compartilhamento de uso dos recursos computacionais como tempo do processador ou CPU, por exemplo, é por *default* proporcional à unidade para cada usuário, sendo partilhado e particionado pelos processos daquele usuário particular. Isto é, se um usuário *User1* submete dois processos P1 e P2 e se outro usuário *User2* submete um único outro processo P3, dada a proporcionalidade de utilização de recursos unitária em relação aos usuários ativos do sistema, teremos que os processos P1 e P2 pertencentes ao primeiro usuário, consumiram juntos a metade do tempo total de processamento enquanto o processo P3 do segundo usuário consumirá sozinho a outra metade, já que é o único processo

daquele usuário *User2*. Proporcionalmente, P1 consome 25%, P2 consome 25% e P3 consome 50%.

5.1.1 Breve histórico dos Grupos de Controle

Os *Process Control Groups* seguem a ideia dos assim denominados *Task Control Groups* [48] e, embora já existissem *patches* do *kernel* modificando o *kernel* 2.6.23 para admitir a existência dos Grupos de Controle, estes *CGroups* foram incorporados ao *kernel* do Linux 2.6.24 como padrão. Enquanto o escalonador ***CFS*** foi concebido para resolver os fundamentais problemas de interatividade e tempo de resposta experimentado pelos usuários de *workstations* e PCs e, baseado em uma divisão proporcional unitária e equânime dos processos garantida pelo ***CFS***, o advento dos *CGroups* atua no sentido de tornar a distribuição de recursos iníqua. Isto é, preserva a proporcionalidade de modo não unitário, permitindo assim que a distribuição dos recursos pelos processos, usuários, grupos de usuários, aplicações, seja feita de maneira proporcional a pesos atribuídos a cada um destes grupos, segundo configurações dos administradores do sistema. Logo, podemos definir em arquivos de configuração, como veremos, Grupos de Controle subordinados aos *CGroups* e formados por usuários, grupos, aplicações, etc do sistema Linux que detêm proporções ponderadas, isto é, pesos distintos ou não de recursos como CPU, I/O, banda de rede, etc.

Os *CGroups* foram escritos como mecanismos de baixo-nível no Linux para containers e máquinas virtuais. Entretanto, não se restringem a máquinas virtuais: eles podem gerenciar os recursos e, portanto, o desempenho de processos ordinários também. Os *CGroups*, ou *Process Control Groups* são partes integrantes do núcleo do *kernel* do Linux e é habilitado em tempo de geração do *kernel*. Oferecem um mecanismo para agrupar processos e controlar vários aspectos de seus comportamentos via vários subsistemas, ou seja, é um componente do *kernel* que é usado para aplicar certas políticas a grupos de processo. Uma hierarquia de *CGroups* [45] [46] [47] [87] é uma estrutura em uma único ponto de montagem com cada subsistema conectado em um diretório pai próprio sob este mesmo ponto de montagem comum. Cada hierarquia apresenta-se ao usuário como um pseudo sistema de arquivo [47] (do tipo "*cgroup*"), no qual cada diretório é um *cgroup* único, contendo em seu interior vários arquivos de controle (alguns pertencentes a subsistemas, e outros a *cgroup* de gerenciamento comum) e *cgroups* descendentes como subdiretórios. Quais subsistemas estão

conectados, é especificado em tempo de criação da hierarquia, que também é um sistema de arquivos em tempo de montagem. Por serem organizados em sistemas de arquivos virtuais, podemos consultar ou configurar *cgroups* lendo ou gravando arquivos. Como exemplo de subsistema, podemos citar o subsistema *cpuset* que permite uma distribuição temporal ou espacial das CPUs, isto é, limita quanto tempo ou fração de tempo de processamento um grupo pode usar ou que CPUs (e bancos de memória para máquinas do tipo NUMA) podem ser usados por um determinado grupo.

Os *CGroups* podem ser usados para melhorar a maneira como o escalonador do Linux distribui o tempo de processamento ou outros recursos para os processos. Ao invés de escalonar o tempo para cada processo, o tempo de processamento é escalonado primeiro no *cgroup,* e para cada porção de tempo que é dada ao *cgroup* é então distribuída para todos os processos que pertencem a este *cgroup*. Os *CGroups* são usados para distribuir todo o tempo de processamento ou outros recursos. A ideia é que *cgroups* são inerentes ao processo pai, tal que todo o processo filho herda o mesmo *cgroup* do pai. Cada processo pertence a um exclusivo *cgroup*.

Os *CGroups* representam os grupos de controle para as tarefas. Se um processo está em um grupo de controle específico ele terá uma porção compartilhada do recurso da máquina. Podemos estabelecer o quão grande ou pequena esta porção pode ser. O que é elegante é que os compartilhamentos são mínimos e não máximos. Se dermos a um grupo 10% de largura de banda de um disco particular e 90% a outro grupo, então se o grupo mais privilegiado não usar todo o seu 90% o grupo menos privilegiado pode usufruir de todo o restante [63].

5.1.2 *CGroups versus* Virtualização

Os *CGroups* foram concebidos como uma forma de virtualização mais leve. Enquanto a virtualização cria uma nova máquina virtual acima da qual o sistema cliente executa, a implementação de *CGroups* funciona criando e delimitando barreiras em torno dos grupos de processos. O resultado é que, enquanto cada cliente virtualizado executa seu próprio *kernel* (e pode executar um diferente sistema operacional do hospedeiro), todos os *CGroups* executam no mesmo hospedeiro. Os *CGroups* perdem pelo completo isolamento fornecido pela solução de virtualização total, mas os *CGroups* tendem a ser mais eficientes.

5.1.3 Formalizando conceitos de *CGroups*

Em conformidade com a documentação do Linux, os termos abaixo são necessários:

- Um *cgroup* associa um conjunto de tarefas ou processos a um conjunto de parâmetros para um ou mais subsistemas.
- Um subsistema é um módulo que faz uso das facilidades de agrupamento fornecida pelos *cgroups* para lidar com grupos de tarefas ou processos de uma maneira particular.
- Uma infraestrutura de *cgroups* oferece apenas a funcionalidade de agrupamento.
- Um subsistema ou controlador de *cgroups* aplica as políticas de contabilização/controle para grupos de tarefas.

Os principais subsistemas ou controladores de *CGroups* e suas respectivas funcionalidades são:

Memory: *Memory controller*

- Permite configurar limites no uso de memória RAM e área de *swap* e efetuar consultas cumulativas do uso por todos os processos

Cpuset: *Cpu set controller*

- Conecta processos dentro de um grupo a conjuntos de CPUs e controla a migração entre CPUs

Cpuacct: *Cpu accounting controller*

- Informação sobre o uso de CPU para um grupo de processos

Cpu: *Cpu scheduler controller*

- Controla a priorização de processos no grupo. Deve-se imaginar como sendo um nível mais avançado do que aquele da priorização dada pelo comando *nice*

Devices: *Devices controller*

- Listas de controle de acesso sobre dispositivos de caractere e de bloco

I/O Controller

- Designa a porção de largura de banda de I/O (baseado no controlador de profundidade da fila)

Network Controller

- Define classes e filas entre a camada de rede genérica e a placa física de rede (NIC). Rotula pacotes com identificador de classes de diferentes prioridades, colocando pacotes com destinos externos em diferentes filas para efeito de tráfego

Freezer

- Este controlador é capaz de congelar um dado *cgroup* tal que cesse o uso de tempo de CPU

5.2 ARQUIVOS DE CONFIGURAÇÃO E DAEMONS DO CGROUP

Os *cgroups* possuem dois arquivos de configuração básicos: */etc/cgconfig.conf* e */etc/cgrules.conf* associados a dois respectivos *daemons* */etc/init.d/cgconfig* e */etc/init.d/cgred*. No arquivo */etc/cgconfig.conf* são definidos e caracterizados os *cgroups* e seus subsistemas ou controladores correlatos com os respectivos parâmetros de controle de cada *cgroup* definido e subsistemas. No arquivo */etc/cgrules.conf* são definidas as associações dos *cgroups* propriamente e os grupos definidos no Linux (usuários, grupos, processos, programas, aplicações e etc.) de modo ligeiramente simples.

Para que os *daemons cgconfig* e *cgred* passem a ficar ativos/inativos devem ser acionados/interrompidos via comando de ativação de serviços e/ou *daemons* do Linux. Isto é,

/etc/init.d/cgred *stop*
/etc/int.d/cgred *start*
/etc/init.d/cgconfig *stop*
/etc/init.d/cgconfig *start*

Ou,

 # service cgconfig *restart*
 # service cgred *restart*

O primeiro *daemon*, *cgconfig*, monta os sistemas de arquivo *cgroups* e cria os *cgroups* com os seus parâmetros corretos. O segundo *daemon*, *cgred*, aplica as regras aos novíssimos processos criados, tais que estes processos obtenham os corretos *cgroups*.

5.3 ADMINISTRANDO E MANTENDO CGROUPS

A manutenção e administração de *cgroups* é relativamente fácil [45] [87] [88], embora haja proposta de criação de uma interface gráfica para gerenciamento de *CGroups*. Devemos considerar as seguintes premissas:

- O *cgroups* são administrados hierarquicamente, os diretórios (*path*) são o nome do *cgroup*, os nomes dos arquivos são configuráveis. Os nomes dos diretórios também podem ser escolhidos.
- *CGroups* filhos estão dentro de seus diretórios pai e valem as noções de herança paterna. A hierarquia pode ser definida pelo administrador.
- O root (o nome *CGroups* vazio ou /) é claramente o ponto de montagem. Existe também alguns grupos *default* do sistema.
- Processos que por ventura não pertençam a nenhum *cgroup* executam no *CGroup* root.
- Existe um *cgroup* para cada tipo de recurso, *CGroups* podem ter mais de um tipo, entretanto é melhor simplificar (ou tipos únicos ou todos os tipos).
- As regras e configurações do *cgroups* aplicam-se a todas as configurações para cada PID que está no *cgroup*.
- A administração do *CGroups* é feita por PID.
- PID filhos de processos pai em um *CGroups* herdam suas características de administração e gerência.
- Acessos a *CGroups* preservam os direitos de acesso e autorizações dos sistemas de arquivos a eles associados, inicialmente apenas o root é permitido alterar *CGroups*.

5.4 EXPERIMENTOS COM *CGROUPS*

Logicamente, só podemos realizar os experimentos abaixo com o *kernel* Linux 2.6.32 ou superior, que incorpora o escalonador ***CFS***. Os experimentos foram divididos com a ausência do *CGroups* e em presença do mesmo. O objetivo desta seção é avaliar o mecanismo e os detalhes de particionamento e compartilhamento dos recursos implementado pelo *CGroups*.

5.4.1 Divisão do tempo de CPU sem *CGroups*

A seguir, será verificado o comportamento para cargas distintas de jobs de usuários diferentes com execução concorrente, ou seja, este experimento pretende verificar a equidade do escalonador para grupos distintos de usuários. São submetidos diferentes jobs com distintas cargas ou número de processos de sorte que, no final das execuções, é verificado o tempo total de cada job por usuários. Espera-se, **sem** CGroups ativo, que sejam tempos de execução muito próximos. Os *CGroups* estão inativos quando os *daemons* ou serviços *cgconfig* e *cgred* estão "parados".

O *kernel*, por *default*, é ativado para CONFIG_USER_SCHED=YES, ou seja, o escalonador ***CFS*** particionará o recurso CPU por ID do usuário. Basicamente, quanto mais processos de um mesmo usuário, menor será a porção de tempo de cada processo do mesmo usuário.

Este experimento consiste em submeter quantidades definidas de processos com método **CF** (carga fixa), através de até 3 jobs de três usuários distintos (IDs diferentes). O objetivo deste experimento é verificar se o escalonador ***CFS*** divide igualitariamente o tempo de CPU por usuário independentemente do número de processos do próprio usuário, quando o *CGroups* está desligado. Ou seja, será verificado se o *default* estabelecido no parágrafo anterior é verdadeiro ou não.

Na Tabela 5.1, as três primeiras colunas identificam a quantidade de processos do tipo **CF** para cada usuário distinto A, B ou C, que submete seu job correspondente, identificados por Job A, Job B e Job C, respectivamente. A quarta e última coluna, contém uma linha para cada resultado do tempo total executado pelo Job X com todos os seus processos correspondentes (100, 200, 300, etc), ou seja, os *n* processos para cada usuário. Se uma das três primeiras colunas estiver vazia, apenas dois jobs dividem CPU. Todos os testes foram executados em *workstation* com dois processadores Intel Xeon Nehalem *Hexacore*.

Job A	Job B	Job C	Tempo Médio (seg)
100	100		188,1739
			188,3847
100	100		189,2279
			188,9056
50	200		231,8728
			234,2683
100	100	100	281,6570
			281,0445
			283,2177
100	200		275,9194
			284,9565
200	200		373,2181
			376,0816
200	200		377,9139
			376,7575
100	300		376,3625
			372,0038
200	300		467,4872
			468,7360
100	400		463,4795
			469,4299
200	300		453,3437
			459,6732
300	200		470,3910
			467,7015
200	400		552,7335
			568,6439
200	200	200	561,3796
			563,4729
			559,8180
300	300		559,8133
			565,1340

Tabela 5-1 Tempos Médios de Execução de processos concorrentes de carga fixada em 10 x 500.000.000 adições unitárias (~ 10 segundos) por 2 ou 3 Jobs de distintos usuários

Os tempos médios da Tabela 5-1, melhor visualizados nos gráficos das Figuras 5-1 e 5-2, mostram uma boa adequação à divisão justa por parte do escalonador ***CFS*** por identificação do usuário, devido à parametrização *default* do *kernel*, independentemente da carga ou número de processos de cada usuário. Pode-se constatar que usuários com cargas idênticas (100 e 100, 200 e 200 e 200, etc) possuem tempos idênticos e usuários com cargas distintas (400 e 200, 100 e 300, etc) também possuem tempos médios de execução próximos. Isto pode ser um problema para usuários com baixa carga concorrendo com outros com alta carga. Mas, o que prevalece é que o escalonador busca se demonstrar justo na partilha aproximadamente igualitária entre os usuários.

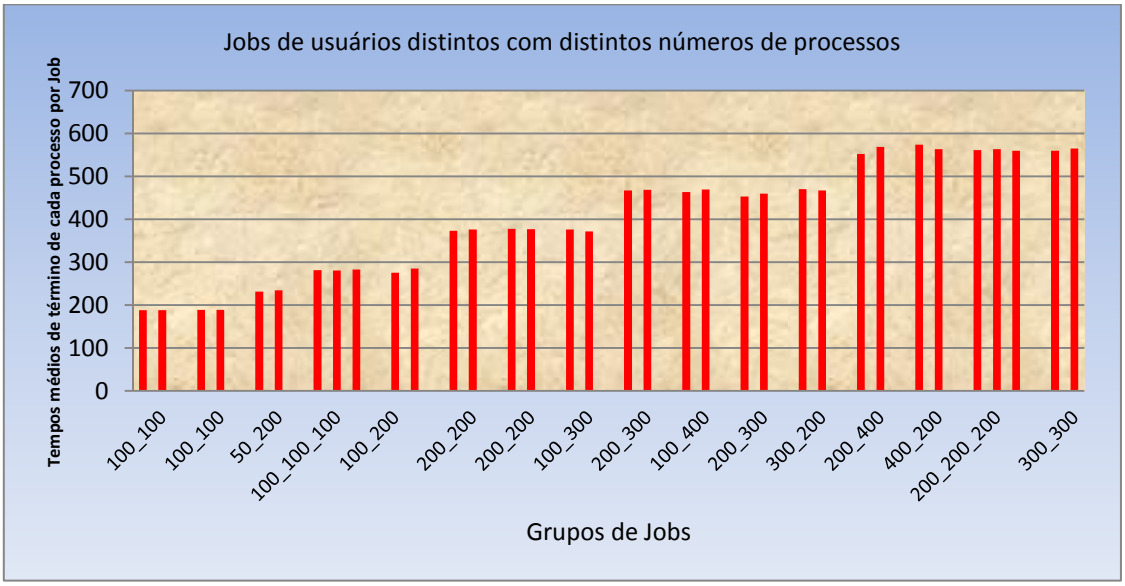

Figura 5-1 Comportamento praticamente equânime para cargas distintas de jobs com diferentes usuários.

Pela análise do gráfico da Figura 5-12, podemos concluir que, **sem** o *CGroups* ativo, por *default* em tempo de compilação de *kernel*, o escalonador ***CFS***, comporta-se objetivando ser equânime para cada usuário. Isto significa que independentemente da quantidade de processos de um mesmo usuário, estes serão escalonados de tal sorte que todos os usuários tenham uma mesma porção, aproximadamente, de tempo de CPU, como pode ser visto quantitativamente na Tabela 5-1.

5.4.2 Divisão de tempo de CPU com *CGroups*

Este experimento consiste em definir dois *cgroups*, um para execução de processos referentes ao grupo de Pesquisadores e outro de Estudantes. O arquivo de configuração é tal que os tempos de CPU proporcionais a cada grupo variam para cada execução, e as quantidades de processos também variam conforme a rodada. Os jobs são repetidos trocando-se as proporções de CPU, ou seja, em uma rodada, por exemplo, são fornecidos para o grupo de Pesquisadores 2048 unidades de CPU e para o de estudantes 1024. Obtém-se os resultados da rodada e executa-se outra com 1024 para o grupo de Pesquisadores e 2048 para o de Estudantes.

N. Procs	Pesquisadores						Estudantes					
	Peso	Média	Mín	Máx	Variância	D. Padrão	Peso	Média	Mín	Máx	Variância	D. Padrão
50	1024	165,2379	156,1979	168,6679	273,9032	3,6764	1024	165,3157	168,5582	156,2124	12,1217	3,4466
100	1024	336,8082	325,3114	340,7924	575,2057	4,4695	1024	334,1319	340,7656	325,4484	18,4840	4,2778
150	1024	500,7522	489,7539	513,1682	838,2764	4,8505	1024	505,7689	513,3686	489,7744	62,4277	7,8747
200	1024	671,9589	654,3952	685,1818	1201,0262	9,6354	1024	669,5881	685,1633	653,8803	76,6921	8,7355
250	1024	838,4950	832,0392	857,8591	1443,1788	6,9243	1024	844,7470	857,8402	832,4972	108,7479	10,4074
200 - Só		333,1445	311,9048	341,1277	68,9992	8,2858						
100	1024	336,3561	273,7003	341,3702	64,3258	7,9801	1024	336,6621	341,5636	307,6372	35,2187	5,9048
100	1024	335,3862	146,8022	342,6916	462,4418	21,3967	2048	337,9948	342,8093	269,6340	93,8312	9,6381
100	1024	333,5649	264,2433	342,0425	111,9677	10,5284	7168	336,7026	341,9709	291,7912	69,7476	8,3096
100	1024	329,3069	266,1116	341,3503	291,4672	16,9868	9216	335,0749	341,3004	306,4817	109,7206	10,4223
100	1024	335,0695	233,2002	342,9110	276,1244	16,5337	19456	335,7730	342,8789	253,7274	243,5617	15,5282
100	2048	340,0875	312,4870	342,6031	16,6321	4,0578	3072	335,5602	342,5679	154,8545	510,5484	22,4821
100	3072	336,6126	190,6012	342,2386	255,4305	15,9021	7168	338,5205	342,3260	282,9065	62,6586	7,8760

Tabela 5-2 Execução de processos concorrentes de 60 segundos cada

Na Tabela 5-2, é confirmado o bom funcionamento do controle de grupos através da parametrização dos arquivos de configuração. Cada linha corresponde a uma execução de dois Jobs de usuários dos grupos Pesquisadores e Estudantes com número de processos de cada Job dado na coluna N. Procs. As colunas seguintes representam, respectivamente para cada grupo: o peso ou fração de CPU, o tempo médio do término dos processos de cada Job, os tempos mínimo, máximo e a variância seguida do desvio padrão para o tempo médio do Job. Pode-se observa que os tempos médios dependem

linearmente do número total de processos quando os pesos ou frações de CPU são idênticos para cada grupo do *CGroups*. Vale ressaltar o caso do Job com 200 processos realizado por um único grupo (Pesquisadores) que é equivalente ao caso de dois Jobs de grupos distintos com 100 processos cada.

Para frações de CPU distintas por grupo, e mesmo número de processos por Job, os tempos médios continuam muito próximos mas as variâncias são bem maiores nestes casos. Isto implica que os processos não tenderam a terminar no mesmo tempo aproximado, ou seja, alguns deles terminarão muito mais cedo e outros muito mais tarde, alargando o intervalo entre o que termina mais cedo e o que termina mais tarde.

5.4.3 Partilha ponderada de recursos compartilhados para *cgroups* ativo

Neste experimento, os dados são extraídos a partir das estatísticas coletadas pelos arquivos de contabilização dos próprios *CGroups* pertencentes aos grupos Pesquisadores e Estudantes, correspondentes ao recurso *cpuacct.* Os usuários A e B pertencem a cada um dos grupos, respectivamente, e suas frações permitidas de CPU são listadas na Tabela 5-3.

Usuário_A Tempo CPU (s)	Usuário_B Tempo CPU (s)	Unid. CpuA	Unid. cpuB	Fração cpuA	Fração cpuB	Razão A/B
59866	60131	1024	1024	0,5000	0,5000	0,9956
40307	79738	1024	2048	0,3333	0,6666	0,5055
30043	90024	1024	3072	0,2500	0,7500	0,3337
24274	95431	1024	4096	0,2000	0,8000	0,2536
15056	128988	1024	7168	0,1250	0,8750	0,1167
12116	107941	1024	9216	0,1000	0,9000	0,1122
7461	112798	1024	15360	0,0625	0,9375	0,0661
6507	113905	1024	19456	0,0500	0,9500	0,0571

Tabela 5-3 Tempos relativos extraídos da contabilização dos *cgroups*

Na Tabela 5-3, a primeira coluna refere-se ao tempo de processamento obtido pelos usuários pertencentes ao grupo de Estudantes (Usuário_A); a segunda, ao de Pesquisadores (Usuário_B). As colunas seguintes, Unid. cpuA e Unid. cpuB, são as proporções de CPU que cabem a cada grupo em um intervalo de tempo. As colunas Fração cpuA e Fração cpuB são os resultados teóricos esperados: frutos das proporções relativas de cada Unidade de CPU dos dois grupos de Estudantes e Pesquisadores. A

última coluna é o resultado experimental obtido após os testes deste experimento, que se revelam bem próximos dos valores teóricos.

Foi realizada uma execução para cada linha da Tabela 5-3 e verifica-se através dos resultados obtidos no gráfico da Figura 5-2 para se constatar a divisão equânime em respeito aos pesos de cada grupo, que representaram os percentuais de recursos alocados.

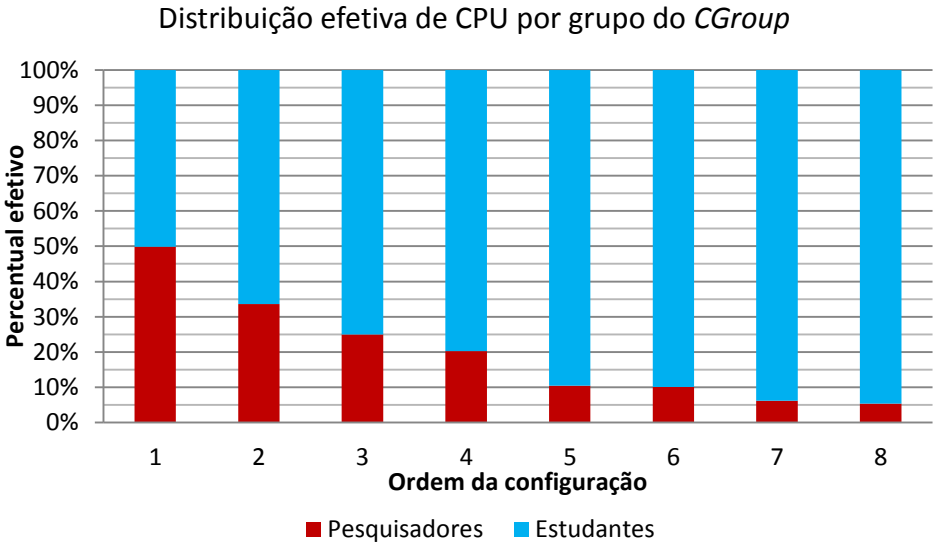

Figura 5-2 Gráfico da distribuição de CPU entre Pesquisadores e Estudantes

Logo, a distribuição dos recursos para os grupos definidos nos *CGroups* comporta-se de modo extremamente satisfatório em conformidade com os objetivos de redefinir a distribuição de recursos de modo ponderado, agora com pesos diferentes da unidade. Em outras palavras, dado que o escalonador ***CFS*** é equânime para todos os processos concorrentes então, através dos *CGroups* podemos subvertê-lo e passar a conferir-lhe distribuição de porções não necessariamente equânimes de recursos, mas proporcionalmente adequadas para a garantia da qualidade de serviço ou outras finalidades que favoreçam certas aplicações em detrimento de outras.

No gráfico da Figura 5-2, podemos verificar a perfeita adequação das proporções de tempo determinadas para cada grupo no arquivo de configuração do subsistema *cpuset* e *cpuacct* conforme definições de proporções de CPU na Tabela 5.3. Observe-se que as proporções: 1024:1024, 1024:2048, . . ., 1024:19456 são preservadas e honradas pela distribuição de tempos de CPU por usuário de cada *CGroup* per si, isto é, proporcionalmente aos seus pesos de CPU ou unidades de CPU para o grupo de usuários A (Estudantes) ou grupo de usuários B (Pesquisadores). Como consequência

direta, temos proporções de utilização de tempo de CPU pelos usuários A e B de aproximadamente 1; 0,5; 0,3; 0,25; 0,11; … ; 0,05, respectivamente. Pode-se concluir diretamente dos dados e informações acima que os *CGroups* efetivamente cumprem a aquilo que propõem: o compartilhamento proporcional dos recursos segundo regras e parâmetros definidos pelo administrador do sistema.

5.4.4 Partilha ponderada de recursos compartilhados para *subgrupos* de *cgroups*

Neste experimento, testamos a possibilidade de definição de particionamento de recursos para subgrupos de *CGroups*. Isto implica na possibilidade de subdividir os recursos dentro de um mesmo grupo do *CGroups*. Por exemplo, em três grupos formados por Pesquisadores, Estudantes e Aplicações, podemos definir novos subgrupos pertencentes ao grupo de Estudantes. Sejam estes subgrupos de Estudantes formados por Graduandos, Mestrandos e Doutorandos. Então, realizam-se duas simulações com estes grupos executando jobs submetidos por cada um deles em um período de 6 minutos (360 segundos) e computaremos através do recurso *cpuacct* o tempo obtido por cada um deles. Duas partilhas de tempo de CPU foram usadas e os relativos a cada uma podem ser observados nas Tabelas 5-4 e 5-5, com seus respectivos gráficos 5-3 e 5-4.

Como um primeiro exemplo de distribuição de CPU para bgrupos de *CGroups*, temos:

GRUPO	*subgrupo*	Fração de CPU	Percentual	Tempo CPU (s)	Percentual Relativo	Percentual de CPU
Aplicações		1024	0,100	36356	0,10138449	0,10138449
Pesquisadores		3072	0,300	108137	0,30155831	0,30155831
Estudantes		6144	0,600	214101	0,59705684	
Estudantes	Graduação	1024	0,125	27301	0,12754139	0,07613455
	Mestrado	3072	0,375	79770	0,37265949	0,22245213
	Doutorado	4096	0,500	106985	0,49979912	0,29834576
	Total do Grupo Estudantes			214056	0,59693244	
	8192					

Tabela 5-4 Dados para teste de subgrupos em *CGroups*

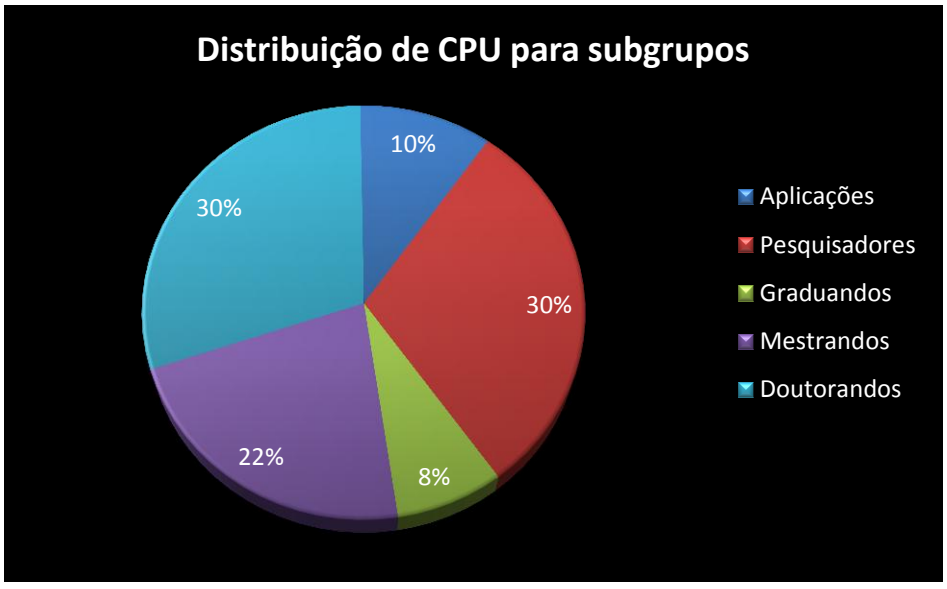

Figura 5-3 Exemplo de partilha de CPU para subgrupos de *CGroups*

Como um segundo exemplo de distribuição de CPU para subgrupos de *CGroups*, temos:

GRUPO	*subgrupo*	Fração de CPU	Percentual	Tempo de CPU (s)	Percentual Relativo	Percentual de CPU
Aplicações		1024	0,125	44767	0,1202	0,1202
Pesquisadores		5120	0,625	239757	0,6430	0,6430
Estudantes		2048	0,250	88020	0,2363	
Total Geral				372544		
Estudantes	Graduação	1024	0,167	15133	0,1719	0,0406
	Mestrado	2048	0,333	29739	0,3379	0,0798
	Doutorado	3072	0,500	43148	0,4902	0,1158
	Total do Grupo Estudantes				0,2363	
	6144			88020		

Tabela 5-5 Um segundo exemplo de distribuição temporal de CPU

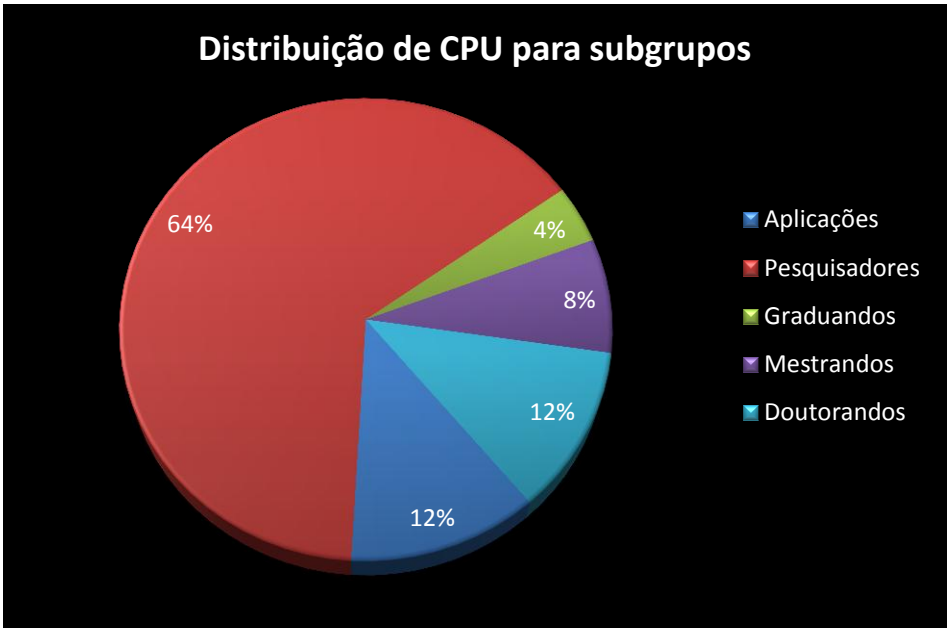

Figura 5-4 Gráfico da segunda distribuição de CPU por subgrupo

Nos resultados apresentados constata-se a justa divisão e contabilização do recurso CPU para grupos e subgrupos definidos segundo critérios de gerenciamento por parte do administrador do sistema para favorecer determinados grupos em detrimento de outros, mas com garantia de serviço, ou seja, respeitando as proporções de CPU a que cada grupo ou subgrupo tem direito. Os gráficos das Figuras 5-3 e 5-4 estão em conformidade com as proporções definidas nas Tabelas 5-4 e 5-5. Os valores não são exatos, mas muito próximos. Isto se deve aos custos de gerenciamento do ***CFS***.

5.4.5 Linearidade do tempo de execução médio para carga fixa (CF) sem *CGroups*

Neste experimento são submetidos vários jobs com o *CGoups* não ativo, isto significa que os serviços ou *daemons cgconfig* e *cgred* estavam inativos ("*stopped*") e deseja-se analisar o comportamento médio de submissões de jobs com vários processos em paralelo de determinadas maneiras. Cada processo de carga fixa (CF) se extingue ao consumir uma carga de 4 segundos de CPU.

Job	N. Procs	Tmin (s)	Tmax (s)	Tmédio (s)	Total Job (s)
1	4	4,31	4,31	4,32	17,28
2	8	8,61	8,68	8,65	69,16
3	12	12,57	12,98	12,88	154,60
4	16	16,59	17,28	17,19	274,97
5	20	21,47	21,63	21,57	431,39
6	24	25,74	25,94	25,88	621,14
7	28	30,09	30,24	30,19	845,21
8	32	34,36	34,54	34,47	1102,92
9	40	39,79	43,21	41,99	1679,72
10	48	48,17	51,84	50,48	2422,99
11	64	68,26	70,33	69,62	4455,90
12	72	75,17	79,03	77,58	5586,05
13	93	101,80	106,82	104,80	9746,51
14	128	138,66	143,76	141,15	18067,15
15	160	175,89	181,42	177,89	28461,98
16	192	210,86	216,35	212,62	40823,67
17	224	246,63	254,36	249,70	55933,08
17	255	279,82	290,05	284,49	72545,53
19	379	415,87	432,33	423,05	160337,38
20	512	562,96	583,81	571,49	292602,00
21	766	849,66	876,09	860,94	659480,44
22	1.024	1128,82	1165,01	1146,20	1173705,90
23	2.048	2234,04	2301,84	2275,76	4660749,00

Tabela 5-6 Tempos Médios de Execução de processos concorrentes de carga fixada em 4 x 500.000.000 adições unitárias por 1 único job de um mesmo usuário

As colunas de 1 a 6 da Tabela 5-6 representam, respectivamente, o número da rodada ou Job, o número de processos concorrentes, o processo que terminou no menor tempo, o processo mais demorado, o tempo médio dos processos e, finalmente, o tempo total cumulativo dos processos daquele Job ou rodada.

Considerando que todos os jobs da Tabela 5-6 foram gerados em uma *mobile workstation* HP Elitebook com processador Intel *quadcore*, podemos verificar que o valor médio é bem comportado para poucos processos por job e estende-se este bom comportamento para muitos processos com cada vez mais um ônus do escalonador como um todo. Nada que não compense a execução paralela dos jobs mas garante aproximadamente uma vazão média linear em função do número de processos e de CPUs ou *cores* disponíveis para processá-los, como verificaremos no gráfico correspondente.

Caso os jobs sejam processados em lotes de processos do tipo 8 + 8, 10 + 6, 4 + 4 + 4 +4, o comportamento é aproximadamente idêntico ao do processamento do total de processos: 16, 16, 16. Para múltiplos jobs com mesma carga de número de processo (1x200, 2x200, 3x200, 4x200, etc.), o tempo médio é praticamente proporcional. Para cargas distintas e com defasagem no tempo, por exemplo, primeiro um job com 200 processos após um tempo determinado de 180 segundos mais um outro job concorrente com 400 processos tem tempo bem diferente do caso contrário. Para concorrência de jobs com cargas idênticas obtemos resultados de tempos médios aproximadamente iguais.

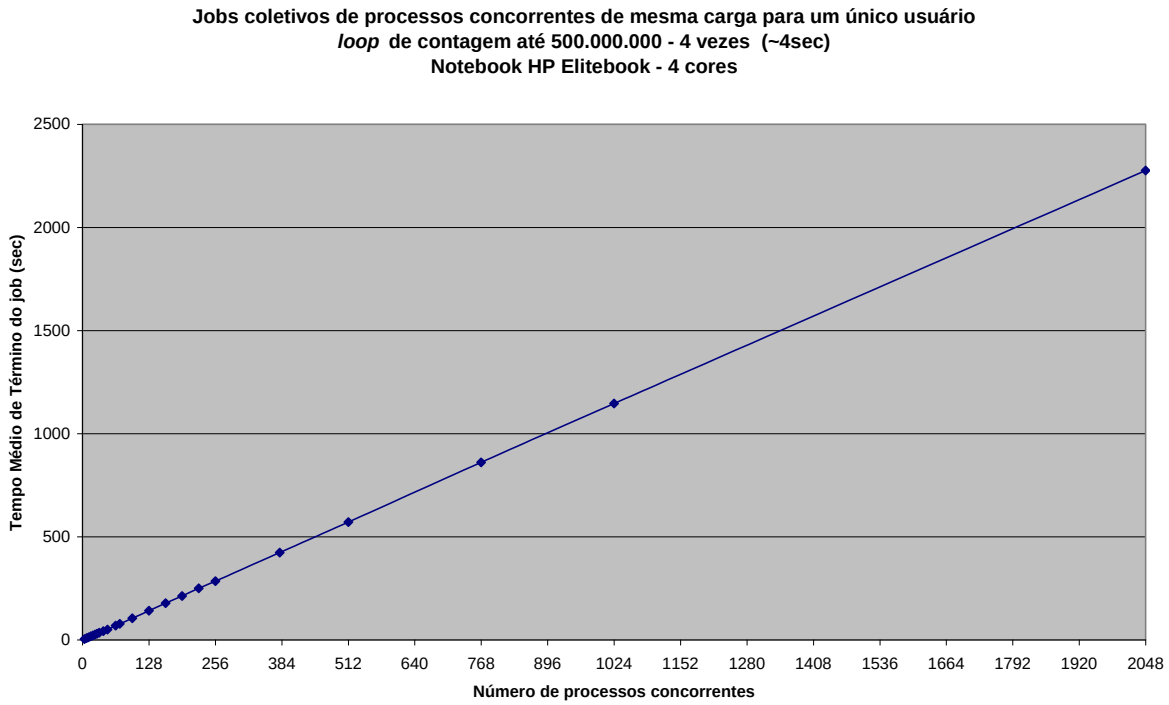

Figura 5-5 Comportamento praticamente linear de Jobs com múltiplos processos concorrentes de mesma carga.

A Figura 5-5 acima mostra o comportamento praticamente linear do escalonador ***CFS*** para executar uma carga fixa e bem definida para diferentes quantidades de processos.

5.5 RESUMO

Neste Capítulo verifica-se formalização do conceito, implementação e exemplos da aplicação de *CGroups* para usuários, grupos de usuários, subgrupos de usuários, aplicações, incorporado no ***CFS*** a partir do kernel 2.6.24 do Linux. São mostrados os principais modos de como compartilhar e definir as formas de compartilhamento destes recursos ou subsistemas e seus respectivos arquivos de configuração dos grupos de controle para o escalonador, bem como os *daemons* a eles associados e como habilitá-los ou desabilitá-los no sistema operacional. Também é exibido a sua estruturação em pseudo arquivo e como administrar e manter os *CGroups*. Nos exemplos deste capítulo, constata-se o bom funcionamento do compartilhamento de recursos através dos *CGroups* e sua facilidade e eficiência de uso. Os resultados demonstram que podemos compartilhar os recursos com pesos definidos pelo administrador do sistema e com distribuição proporcional a estes pesos.

CAPÍTULO 6 – CONCLUSÕES E TRABALHOS FUTUROS

O escalonador ***CFS*** é mais justo que seu antecessor, ***O(1)***, e melhora o tempo de resposta experimentado pelo usuário para aplicações interativas causando uma melhor impressão de desempenho por parte do usuário de *workstation* ou PC, principalmente.

O ***CFS*** do *kernel* do Linux é dinamicamente reconfigurável via interface de linha de comando ***sysctl***. Dentre elas, destacam-se: *kernel.sched_latency_ns*, *kernel.sched_min_granularity_ns* e *kernel.sched_wakeup_granularity_ns*, que são alguns dos parâmetros de configuração, fundamentais para adequação do escalonador a aplicações específicas como as direcionadas a processamento CPU intensivo permitindo reduzir o número de trocas de contexto e favorecendo as aplicações CPU intensiva. Estes parâmetros podem ser configurados dinamicamente via comando *sysctl* trocando-se os valores *default* pelos adequados às aplicações específicas do usuário em unidades de nanossegundos.

O ***CFS*** possui o conceito de *vruntime* e um algoritmo de heurística simples, para controlar e equilibrar a divisão do tempo de processamento por processo por CPU, calculado como a média aritmética entre os tempos gastos por cada processo em cada fila de execução. As filas de execução, por sua vez, são representadas em estruturas formadas por árvores binárias rubro-negras. Estas árvores possuem complexidade computacional de $O(\log n)$ para inserções e atualizações de novos nós, onde n é o número de nós da árvore, em que cada nó representa um processo.

A granularidade de tempo do ***CFS*** passa a ter a precisão mensurável de nanossegundo, e é obtida graças às rotinas da biblioteca *hrtime* (*high resolution time*) implementadas no *kernel*, desde a versão 2.6.21. Esta granularidade de tempo mais fina e precisa permite que o novo escalonador ***CFS*** possa distribuir porções tão pequenas ou tão grandes quanto for necessário para as aplicações, transferindo para o administrador do sistema a difícil tarefa de decidir o melhor para as suas aplicações, ao invés dele, escalonador, resolver o melhor para todas sem conhecê-las.

O novo escalonador disponibiliza através da biblioteca *libcgroups* os grupos de controle (*CGroups*), que permitem o compartilhamentos e a distribuição dos recursos como CPU, memória, largura de banda, etc, de modo ponderado com referência a pesos definidos pelo administrador do sistema para a partilha e/ou compartilhamento dos recursos. Isto permite traçar perfis de usuários e aplicações, concedendo a possibilidade de definição de garantia de serviço, ou seja, um ***QoS*** para aplicações e/ou usuários ou

grupos de usuários, permitindo assim a implementação de ***SLAs***. A garantia desses Acordos de Nível de Serviços é fundamental para a credibilidade e confiabilidade dos novos modelos de negócios. Esta nova característica do escalonador do Linux é de crucial importância para estes modelos de negócios de cuja cobrança dependem as tarifações e contabilizações dos recursos efetivamente utilizados e não injustamente pagas pela alocação de recurso e não necessariamente pelo seu uso efetivo como CPU, I/O, largura de banda, etc. Os serviços disponibilizáveis pela Computação em Nuvem, com seus XaaS (onde: X=*Software*, *Infrastructure*, *Storage*, *Platform*, etc), podem servir como um bom exemplo desse modelo de negócio *pay-per-use*.

Estão disponíveis ferramentas ou utilitários para diagnóstico, depuração e estatísticas do escalonador e dos recursos do sistema em *pseudo-file system*: *sched_debug* e *schedstat* de grande aplicabilidade para análise e gerenciamento mais refinado do *kernel* do Linux e que foram as ferramentas básicas deste trabalho para, dentre outros, verificar o balanceamento de carga, a distribuição espacial de CPU, a contabilização de CPU alocada por processador, a medição do número de trocas de contexto.

É possível diminuir drasticamente o número de trocas de contexto, objetivando um maior desempenho dos processos no sistema através da mudança dinâmica dos parâmetros do escalonador do *kernel*. Simplesmente, pode-se imaginar que quanto maior for a granularidade mínima, menor será o número de trocas de contexto por unidade de tempo. Isto é muito conveniente para aplicações do tipo HPC, que, em geral, são CPU intensivas, e diminuir a troca de contexto implica em disponibilizar mais CPU para as aplicações, aumentando assim a eficiência e otimização do sistema como um todo, satisfazendo melhor os objetivos do escalonador e as necessidades das aplicações e dos usuários do sistema: maior eficiência, maior *throughput*, menor tempo de resposta, etc.

Como trabalhos futuros, propomos um escalonador específico voltado para o ambiente HPC que aproveite as características do ***CFS*** e do *CGroups*; o projeto de um *framework* que aproveite as características do novo escalonador para adequar aplicações com determinadas características em sintonia com as devidas parametrizações do escalonador ***CFS*** que favoreçam a essas aplicações; testar com *CGroups* o comportamento do compartilhamento de outros recursos como memória, largura de banda, etc; a adoção de *kernel tickless* para economia de energia em estados ociosos do sistema, como experimento apresentado no Apêndice A, pois os ganhos em escala

seriam enormes, devido a quantidade de dispositivos embarcados, jogos, *tablets*, celulares de última geração, *notebooks*,*ultrabooks*, que tendem a aumentar em volume de vendas (maiores do ponto de vista do usuário final, que subutiliza seu equipamento mas consomem muita energia desnecessariamente), ou mesmo empresas, instituições acadêmicas, etc.

REFERÊNCIAS

[1] A. Silberschatz, G. Gagne e P. B. Galvin, "*Operating System Concepts*," 2005. [Online]. Disponível em: 7o ed. John Wiley & Sons. Inc, 2005, pp. 940.

[2] Netmarketshare, "*Marketshare Operating Systems*," 2006. [Online]. Disponível em: http://www.netmarketshare.com/operating-system-market-share. [Acesso em 01 05 2012].

[3] T. Statistics, "*Operating System Family System Share*," Jun 2012. [Online]. Disponível em: http://www.top500.org. [Acesso em 02 07 2012].

[4] B. Jernej, "*Multi-core architectures*," [Online]. Disponível em: http://www-bcf.usc.edu/~jbarbic/multi-core-15213-sp07.ppt. [Acesso em 11 10 2011].

[5] N. Karmakar, "*Multi-core Architecture*," Computer Science & Engineering, North Maharashrta University, India., HPCC '10 Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications, Pages 573-578, IEEE Computer Society Washington, DC, USA ©2010, 2010. [Online]. Disponível em: http://www.kernel.org/doc/ols/2001/elss.pdf. [Acesso em 16 12 2011].

[6] B. Schauer, "*Multicore Processors - A Necessity*," Set 2008. [Online]. Disponível em: http://www.csa.com/discoveryguides/multicore/review.pdf. [Acesso em 13 03 2012].

[7] S. Siddha, "*Multi-core and Linux Kernel*," Intel Open Source Technology Center, 2007, DEBS'2010 – In Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, pp. 235-247. [Online].

[8] S. Siddha, V. Pallipadi e A. Mallick, "*Process Scheduling Challenges in the Era of Multi-Core Processors*," Intel Technology Journal, vol. 11, no. 04, p. 361-370, nov- 2007., Nov 2007. [Online]. Disponível em: http://download.intel.com/technology/itj/2007/v11i4/9-process/9-Process_Scheduling_Challenges.pdf. [Acesso em 23 12 2011].

[9] R. R. M. U. e. K. L. S. Balakrishnan, "*The Impact of Performance Asymmetry in Emerging Multicore Architectures,"* In the Proceedings of the 32th International Symposium on Computer Architecture, ISCA'2005, 2005, [Online]. Disponível

em: http://pages.cs.wisc.edu/~saisanth/papers/isca05.pdf. [Acesso em 14 01 2012].

[10] W. Jiang, Y. S. Zhou, Y. Cui, W. Feng, Y. Chen, Y. C. Shi e Q. B. Wu, "*CFS Optimizations to KVM Threads on Multi-Core Environment,*" In Proceedings of the 15th International Conference on Parallel and Distributed Systems of the IEEE ICPADS, Shenzhen, 8-11 December 2009, pp. 348-354. doi:10.1109/ICPADS.2009.83, Dec 2009. [Online].

[11] Wikipedia, "*Usage share of operating systems,*" [Online]. Disponível em: http://en.wikipedia.org/wiki/Usage_share_of_operating_systems. [Acesso em 10 07 2011].

[12] R. Brenila, "*História das Interfaces Gráficas de Usuário (GUIs),*" 05 12 2005. [Online]. Disponível em: http://www.vivaolinux.com.br/imagens/dicas/comunidade/Historia_GUIs.pdf. [Acesso em 29 01 2012].

[13] A. Mills, "*Interview with Con Kolivas, part 1: computing is boring,*" 24 07 2007. [Online]. Disponível em: http://apcmag.com/interview_with_con_kolivas_part_1_computing_is_boring.htm . [Acesso em 03 03 2012].

[14] A. Mills, "*Interview with Con Kolivas, part 2: his effort to improve Linux performance on the desktop,*" 24 07 2007. [Online]. Disponível em: http://apcmag.com/interview_with_con_kolivas_part_2_his_effort_to_improve_linux_performance_on_the_desktop.htm. [Acesso em 03 03 2012].

[15] R. M. Stallman, "*Software Livre,*" [Online]. Disponível em: http://pt.wikipedia.org/wiki/Projeto_GNU. [Acesso em 14 05 2012].

[16] R. M. Stallman, "*Projeto GNU,*" [Online]. Disponível em: http://pt.wikipedia.org/wiki/Richard_Matthew_Stallman. [Acesso em 14 05 2012].

[17] A. S. Tanenbaum, "*Modern Operating System*", 3 ed., Pearson/Prentice Hall, 2008, p. 1104.

[18] O'Reilly, "*Open Sources: Voices from the Open Source Revolution,*" 1o ed. O'Reilly, 1999. Appendix A – "*The Tanenbaum-Torvalds Debate*" , Jan 1999. [Online]. Disponível em: http://oreilly.com/catalog/opensources/book/appa.html. [Acesso em 04 03 2012].

[19] L. Torvalds, "*FM Interview with Linux Torvalds: "What motivates free software developers?",*" retrieved from the World Wide Web, 14 December 2001, 1998. [Online]. Disponível em: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/583/504. [Acesso em 30 01 2012].

[20] L. Torvalds e D. Diamond, "*Just for Fun: the Story of an Accidental Revolutionary*", Reprint ed., New York, NY: Harper Business, 2001, p. 288.

[21] M. T. Jones, "*Inside the Linux scheduler,*" 30 06 2006. [Online]. Disponível em: http://www.ibm.com/developerworks/linux/library/l-scheduler/. [Acesso em 21 02 2012].

[22] M. T. Jones, "*Inside the Linux 2.6 Completely Fair Scheduler,*" Dec 2009. [Online]. Disponível em: http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler. [Acesso em 17 05 2012].

[23] J. Corbet, "*The Rotate Staircase Deadline Scheduler,*" 06 03 2007. [Online]. Disponível em: http://lwn.net/Articles/224865/. [Acesso em 05 05 2012].

[24] C. Kolivas, "*RSDL completely fair starvation free interactive cpu scheduler,*" [Online]. Disponível em: http://lwn.net/Articles/224654/ . [Acesso em 05 05 2012].

[25] J. Kobus e R. Szklarski, "*Completely Fair Scheduler and its tuning,*" 09 09 2011. [Online]. Disponível em: http://pdf.aminer.org/000/566/986/operating_system_multilevel_load_balancing.pdf. [Acesso em 27 03 2012].

[26] T. M. Le, "*A Study on Linux Kernel Scheduler Version 2.6.32,*" [Online]. Disponível em: http://pt.scribd.com/doc/24111564/Project-Linux-Scheduler-2-6-32. [Acesso em 03 06 2012].

[27] C. S. Wong, I. K. T. Tan, R. D. Kumari, J. W. Lam e W. Fun, "*Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers,*" IEEE, 2008, Information Technology, 2008, ITSim 2008. International Symposium on, 26-28 Aug. 2008, Vol 4, pp. 1-8, Aug 2008. [Online].

[28] C., S. Wong, I. Tan, R. D. Kumar e F. Wey, "*Towards Achieving Fairness in the Linux Scheduler,*" SIGOPS Operating Systems Review, vol. 42, no. 5, Jul. 2008, pp.34-43, Jul 2008. [Online]. Disponível em:

http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf. [Acesso em 18 03 2012].

[29] T. Gleixner, "*The Completely Fair Scheduler,*" Japan, Jul 2008. [Online]. Disponível em: http://www.linuxfoundation.jp/jp_uploads/seminar20080709/lfjp2008.pdf. [Acesso em 19 04 2012].

[30] M. Hailperin, "*Changes in the Linux Scheduler as of 2.6.23,*" in Operating Systems and Middleware: Supporting Controlled Interaction, [Online]. Disponível em: https://gustavus.edu/+max/osbook/updates/CFS.html. [Acesso em 10 07 2011].

[31] I. Molnar, "*This is CFS scheduler,*" [Online]. Disponível em: http://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt. [Acesso em 04 08 2011].

[32] A. Kumar, "*Multiprocessing with the completely fair scheduler,*" 2008 01 08. [Online]. Disponível em: http://www.ibm.com/developerworks/linux/library/l-cfs/. [Acesso em 25 12 2011].

[33] P. Turner, B. B. Rao e N. Rao, "*CPU bandwidth control for CFS,*" The Proceedings of the Linux Symposium, Ottawa, Ontario - Canada, 2010, pp. 245-254, 2010. [Online]. Disponível em: http://www.kernel.org/doc/ols/2010/ols2010-pages-245-254.pdf. [Acesso em 11 12 2011].

[34] K. Salah, A. Manea, S. Zeadally e J. M. A. Calero, "*On Linux Starvation of CPU-bound processes in the presence of network I/O,*" Computer and Electrical Engineering, 04 07 2011. [Online].

[35] M. R. Nikseresht, A. Somayaji e A. Maheshwari, "*Customer Appeasement Scheduling,*" Tech. Report 10-18, School of Computer Science, Carleton University, 15 12 2010. [Online]. Disponível em: http://arxiv.org/pdf/1012.3452v1. [Acesso em 03 01 2012].

[36] R. Bayer, "*Symmetric binary B-Trees: Data structure and maintenance algorithms,*" Acta Informatica, vol. 1, pp. 290-306, 1972, 1972. [Online].

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, "*Introduction to Algorithm,*" 3o ed. Cambridge, Massachusetts: The MIT Press, 2009, pp. 1312, 2009. [Online]. Disponível em: 3o ed. Cambridge, Massachusetts: The MIT Press,

2009, pp. 1312.

[38] J. L. Szwarcfiter e L. Markenzon, "*Estrutura de Dados e seus Algoritmos*", 3a. ed., Rio de Janeiro: LTC, 2010.

[39] S. Seelam, R. Romero, W. Buros e P. Teller, "*Enhancements to linux I/O scheduling,*" [Online]. Disponível em: http://www.kernel.org/doc/ols/2005/ols2005v2-pages-183-200.pdf. [Acesso em 05 05 2012].

[40] A. Gud, "*Linux O(1) CPU Scheduler,*" 2005 04 27. [Online]. Disponível em: http://people.cis.ksu.edu/~gud/docs/ppt/scheduler.pdf. [Acesso em 01 06 2011].

[41] M. T. Jones, "*Anatomy of Linux Kernel,*" 07 06 2007. [Online]. Disponível em: http://www.ibm.com/developerworks/br/library/l-linux-kernel/#icomments. [Acesso em 17 05 2012].

[42] M. T. Jones, "*Linux and Symmetric Multiprocessing,*" Report: IBM DeveloperWorks, Mar 2007, Dec 2007. [Online]. Disponível em: http://www.ibm.com/developerworks/library/l-linux-smp. [Acesso em 30 04 2012].

[43] P. Turner, "*Google Improving 'global' scheduler decisions,*" 2011. [Online]. Disponível em: http://linuxplumbersconf.net/2011/ocw//system/presentations/309/original/LPC_2011_Global_Lag.pdf . [Acesso em 17 05 2012].

[44] A. Abbas, "*Linux CFS Algorithm and Virtual Runtime,*" O'Reilly Media; Third Edition edition (November 2005), pp. 944, Fev 2012. [Online]. Disponível em: http://alouche.net/2012/02/04/linux-cfs-algorithm-and-virtual-runtime/ . [Acesso em 21 08 2011].

[45] S. Ellis, "*Resource Management with CGroups,*" Brisbane - Australia, 2011. [Online]. Disponível em: http://sysadmin.miniconf.org/2011/lca2011-sysadmin-miniconf_steven-ellis_CGroups.pdf. [Acesso em 05 05 2012].

[46] K. Hiroyu, "*Cgroup And Memory Resource Controller,*" Japan Linux Simposium, 19 11 2008. [Online]. Disponível em: , http://www.linuxfoundation.jp/jp_uploads/seminar20081119/CgroupMemcgMaster.pdf. [Acesso em 03 03 2012].

[47] R. Oxenham, "*Resource Management CGroup Lab,*" Red Hat EMEA, 2011.

[Online]. Disponível em: http://emea-partner-summit.com/system/files/other-files/dublin/cgroup-lab.pdf. [Acesso em 25 04 2012].

[48] A. A. S. Pereira, "*Task Control Group,*" 2006. [Online]. Disponível em: http://www.lsd.ic.unicamp.br/mo806wiki/index.php/Task_Control_Groups. [Acesso em 05 05 2012].

[49] M. Prpic, R. Landmann e D. Silas, "*Resource Management Guide,*" [Online]. Disponível em: http://docs.fedoraproject.org/enUS/Fedora/15/pdf/Resource_Management_Guide/Fedora-15-Resource_Management_Guide-en-US.pdf. [Acesso em 05 05 2012].

[50] C. Williams, "*Linux Scheduler Latency,*" Red Hat Inc, mar-2002, [Online]. Disponível em: http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf. [Acesso em 12 12 2011].

[51] M. R. Garey e D. S. Johnson, "*Computers and Intractability: A Guide to the Theory of NP-Competeness*", 1a ed., W. H. Freeman, 1979, p. 340.

[52] O. Sinnen, "*Task Scheduling of Parallel Systems*", New Zealand: Wiley-Interscience, 2007, p. 296.

[53] J. D. Ullman, "*NP-complete scheduling problem,*" Jun 1976. [Online]. Disponível em: Journal of Computer and System Sciences, vol. 10, no. 3, pp. 384-393.

[54] D. P. Bovet e M. Cesati., "*Understanding the Linux Kernel,*" O'Reilly Media; Third Edition edition (November 2005), pp. 944. [Online].

[55] R. Love, "*Linux Kernel Development,*" 3o ed. Addison-Wesley, 2010, pp. 440, 2010. [Online].

[56] C. Martins, R. Peixoto e R. Faveri, "*Process Scheduling,*" [Online]. Disponível em: http://www.ic.unicamp.br/~islene/2s2011-mo806/escalonador/cristianno-robson-rodrigo.pdf. [Acesso em 12 04 2012].

[57] W. Mauerer, "*Professional Linux Kernel Architecture,*" Wiley Publishing Inc., 2008, 1368 pp., 2008. [Online]. Disponível em: Wiley Publishing Inc., 2008, 1368 pp..

[58] C. Deives e P. Sabóia, "*Escalonamento no Linux,*" Dec 2007. [Online]. Disponível em: http://www.ic.unicamp.br/~islene/2s2007=mo806/slides/MO806_Escalonamento_no_Lin.html. [Acesso em 19 02 2012].

[59] M. Kravetz, H. Franke, S. Nagar e R. Ravindran, "*Enhancing Linux Scheduler Scalability,*" in the 2001 Ottawa Linux Symposium, Jul 2001. [Online]. Disponível em: http://www.kernel.org/doc/ols/2001/elss.pdf. [Acesso em 04 10 2011].

[60] S. P. Molloy e P. Honeyman, "*Scalable Linux Scheduling,*" Technical Report 01-7, University os Michigan, 01-maio-2001, 01 05 2011. [Online]. Disponível em: http://www.citi.umich.edu/techreports/reports/citi-tr-01-7.pdf. [Acesso em 05 05 2012].

[61] W. Ma e J. Wang, "*Analysis of the Linux 2.6 kernel scheduler,*" International Conference On Computer Design And Applications, 2010, (ICCDA2010), Qinhuangdao, China, 25 - 27 June 2010, 2010. [Online]. Disponível em: International Conference On Computer Design And Applications, 2010, (ICCDA2010), Qinhuangdao, China, 25 - 27 June 2010.

[62] D. M. Madhavrao, "*Linux Process Management,*" MTech - 1st Year, School of Information Technology, IIT Khargpur, [Online]. Disponível em: http://sit.iitkgp.ernet.in/research/aut05seminar2/report4.doc. [Acesso em 16 01 2012].

[63] J. Aas, "*Understanding the Linux 2.6.8.1 CPU Scheduler,*" Silicon Graphics Inc. (SGI), 2005. [Online]. Disponível em: http://joshaas.net/linux/linux cpu scheduler.pdf. [Acesso em 22 06 2011].

[64] A. Kumar, "*Introducing the CFS for Linux Multiprocessing with the Completely Fair Scheduler,*" Jan 2008. [Online]. Disponível em: http://www.ibm.com/developerworks/linux/library/l-cfs/ . [Acesso em 27 08 2011].

[65] C. S. Pabla, "*Completely Fair Scheduler,*" 2009. [Online]. Disponível em: http://www.linuxjournal.com/article/10267 . [Acesso em 12 09 2011].

[66] Jeremy, "*Linux: Linus On CFS vs SD,*" 27 07 2007. [Online]. Disponível em: http://ryanorser.com/2007/07/ . [Acesso em 29 05 2012].

[67] J. Corbet, "*A new approach to kernel timers,*" 20 07 2005. [Online]. Disponível em: http://lwn.net/Articles/152436/ . [Acesso em 16 08 2011].

[68] J. Corbet, "*The High Resolution Timer API,*" 16 01 2006. [Online]. Disponível em: http://lwn.net/Articles/167897/ . [Acesso em 02 01 2012].

[69] T. Gleixner, "*ktimers subsystem,*" Jan 2006. [Online]. Disponível em: http://article.gmane.org/gmane.linux.kernel/333393. [Acesso em 11 10 2011].

[70] T. Gleixner e D. Niehaus, "*Hrtimers and Beyond,*" 2006, In Proc. Linux Symposium, Ottawa, Ontario, June, vol1, pp. 333-346, 2006, 2006, http, 2006. [Online]. Disponível em: http://kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf. [Acesso em 12 02 2012].

[71] H. A.-W. I. Stoica, K. Jeffay, S. K. Baruah, J. E. Gehrke e C. G. Plaxton, "*A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems,*" Real-Time Systems Symposium, 1996., 17th IEEE,04/jun/Dec,1996, pp 288-299, 1966. [Online]. Disponível em: http://www.cs.unc.edu/~jeffay/papers/RTSS-96a.pdf. [Acesso em 15 01 2012].

[72] J. Regehr, "*Some Guidelines for Proportional Share CPU Scheduling in General-Purpose Operating Systems,*" [Online]. Disponível em: http://content.lib.utah.edu/cdm/singleitem/collection/uspace/id/59. [Acesso em 03 09 2011].

[73] L. Jelenkovic, S. Gros e D. Jakobovic, "*Testing Task Schedulers on Linux System,*" In Proceedings of SCSS. 2009,pp. 385-390, 2009. [Online].

[74] T. Groves, J. Knockel e E. Schulte, "*BFS vs. CFS - Scheduler Comparison,*" 11 Dec 2009. [Online]. Disponível em: http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs-groves-knockel-schulte.pdf. [Acesso em 12 02 2012].

[75] L. A. Torrey, J. Coleman e B. P. Miller, "*A comparison of interactivity in the Linux scheduler and an MLFQ scheduler,*" Journal Software - Practice & Experience, vol. 37 Issue 4, April 2007, pp. 347-364, John Wiley & Sons, Inc. New York, NY, USA, 24 10 2006. [Online]. Disponível em: http://onlinelibrary.wiley.com/doi/10.1002/spe.772/pdf. [Acesso em 10 06 2012].

[76] T. M. e. F. V. N. G. Anderson, "*Application of Global and One-Dimensional Local Optimization to Operating System Scheduler Tuning,*" [Online]. Disponível em: http://www.prasa.org/proceedings/2010/prasa2010-02.pdf . [Acesso em 29 09 2011].

[77] M. Hailperin, "*Operating Systems and Middleware: Supporting Controlled Interaction,*" 1a. edition, pp. 496, [Online]. Disponível em:

https://gustavus.edu/+max/os-book/free-osbook.pdf. [Acesso em 30 12 2011].

[78] M. Bligh, M. Desnoyers e R. Schultz, "*Linux Kernel Debugging on Google-sized clusters,*" 2007. [Online]. Disponível em: http://www.kernel.org/doc/ols/2007/ols/2007v1-pages-29-40.pdf. [Acesso em 10 01 2012].

[79] Principled Technology, Inc., "*Performance comparision of three consumer notebooks PCs with Intel and AMD processors - iPARK test and setup procedure,*" Aug 2006. [Online]. Disponível em: http://www.principledtechnologies.com/clients/reports/intel/consnoteperf0806.pdf. [Acesso em 08 05 2012].

[80] P. J. Salzman, M. Burian e O. Pomerantza, "*The Linux Kernel Module Programming Guide,*" 2001, 2007-05-18 ver 2.6.4, 18 05 2007. [Online]. Disponível em: http://tldp.org/LDP/lkmpg/2.6/html/ . [Acesso em 02 01 2012].

[81] C. Li, C. Ding e K. Shen, "*Quantifying The Cost of Context Switch,*" Workshop on Experimental Computer Science, 2007. [Online]. Disponível em: http://www.cs.rochester.edu/u/cli/research/switch.pdf. [Acesso em 20 06 2012].

[82] F. Liu, F. Guo, Y. Solihin, S. Kim e A. Eker, "*Characterizing and modeling the behavior of context switch misses,*" [Online]. Disponível em: In Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 91–101, 2008.

[83] J. Sjöblom, "*Power Efficiet Scheduling for a Cloud System,*" Master of Science Thesis, Department of Information Technology - Abo Akademi University, 2011. [Online]. Disponível em: http://www.cloudsoftwareprogram.org/theses-and-articles/i/27888/1570/master-s-thesis-power-efficient-scheduling-for-a-cloud-system-Joachim-Sjoblom.pdf. [Acesso em 13 03 2012].

[84] C. Rajan, "*Analyzing the impact os sysctl scheduler tunables,*" [Online]. Disponível em: http://events.linuxfoundation.org/slides/2011/linuxcon/lcna2011_rajan.pdf. [Acesso em 07 01 2012].

[85] O. Sukwong e H. S. Kim, "*Is Co-Scheduling Too Expansive for SMP VM's,*" 10 04 2011. [Online]. Disponível em: EuroSys'11, Salzburg - Austria, 10-abr-2011, In Proceedings of the ACM European Conference on Computer systems, 2011,

http://theone.ece.cmu.edu/papers/96.coscheduling.2011eurosys.pdf. , [Acesso em 01 04 2012].

[86] S. Siddha, V. Pallipadi e A. V. D. Ven, "*Getting maximum mileage out of tickless,*" In Proceedings of the Linux Symposium, Ottawa, Ontario - Canada, 2010, pp. 201-208, 2010. [Online]. Disponível em: http://kernel.org/doc/ols/2007/ols2007v2-pages-201-208.pdf. [Acesso em 30 03 2012].

[87] Y. Georgiou e M. Hautreux, "*SLURM Resources isolation through groups,*" Sep 2011. [Online]. Disponível em: http://www.schedmd.com/slurmdocs/slurm_ug_2011/SLURM_UserGroup2011_cgroups.pdf. [Acesso em 05 05 2012].

[88] S. Seyfried, "*Resource Management in Linux with Control Groups,*" Linux-Kongress 2010: 2010, 2010. [Online]. Disponível em: http://www.linux-kongress.org/2010/slides/seyfried-cgroups-linux-kongress-2010-presentation.pdf. [Acesso em 01 04 2012].

Apêndice A – DO SISTEMA OPERACIONAL SINCOPADO AO SISTEMA OPERACIONAL RITIMADO

No passado, o *kernel* do Linux interrompia cada CPU com uma frequência pré-determinada. Esta periodicidade correspondia a frequências de 100 Hz, 250 Hz ou 1000 Hz, dependendo da plataforma, ou seja, interrupções em cada CPU de 100, 250 ou 1000 vezes por segundo. Esta frequência, definida na variável de ambiente HZ ou NO_HZ, por exemplo, era configurável em tempo de compilação de *kernel*. O *kernel* consultava cada CPU sobre o processo que ele estava executado, e usava os resultados para o processo de contabilização e balanceamento de carga. Conhecido como demarcação ou marcação do temporizador, o *kernel* realizava esta interrupção, independentemente do estado de energia da CPU. Portanto, mesmo uma CPU ociosa estava respondendo a até 1000 destes pedidos a cada segundo. Em sistemas que implementam medidas de economia de energia para CPUs ociosas, a marcação do temporizador impõe que a(s) CPU(s) permaneçam inativas em estados mais baixos de energia por tempo suficiente para que o sistema se beneficie desta economia de energia .

Um *kernel tickless* ou *dyntick* ou NO_HZ é aquele no qual o sistema substitui as antigas interrupções periódicas dos temporizadores, por interrupções sob demanda. Isto significa que àas CPUs ociosas é permitido permanecer ociosas até que novos processos sejam enfileirados para execução, e as CPUs que entram nos estados de mais baixa energia neles permanecem por longos períodos.

Uma das principais vantagens do sistema operacional *tickless*, *dyntick* ou NO_HZ é a economia de energia [86] pois evita que o escalonador interfira em tempos demarcados pelo *tick*, melhorando assim a economia de energia, como podemos constatar nos gráficos abaixo, obtidos com medição direta de dispositivo Intel para consumo de energia de equipamentos de informática. O dispositivo de medição da Intel utilizado para este experimento denomina-se iPARK (*Intel Power Aquisition Recording Kit*) Rev_1.0.081104, desenvolvido pela empresa *SSG.CSD.AET.Power Tracing Technologies* [79].

Foram realizadas quatro medições em dois equipamentos físicos, cada um deles com o sistema operacional Linux *Kernel* 2.6.19 (com *tick* HZ = 1000 Hz) e com *Kernel* 2.6.32 (*tickless*). Todas as duas workstations foram submetidas ao testes, uma para cada

kernel, antigo ou novo (*tickless*). A economia foi de 18,09% no caso da workstation HP Z800 e de 12,83% para o HP Proliant 380. A relevância deste experimento é constatar que se pode economizar a bateria dos PCs ou notebook de usuários comuns apenas instalando o Linux com *kernel tickless*. Isto economiza, como mostram os gráficos das Figuras A-1 a A-4, de 10% a 15% de energia para CPUs ociosas e estes nossos equipamentos, em geral, ficam mais ociosos do que ativos. Portanto, o ganho de economia de energia é líquido e certo.

Outros ambientes que podem tirar um enorme proveito são aqueles em que os equipamentos ficam ligados 24 horas por dia como no mundo acadêmico, por exemplo. Além destes, os ambientes de HPC também, por uma questão de escala, podem se beneficiar do *kernel tickless*. É bom atentar para o detalhe que a economia é em função do tempo ocioso do sistema (*idle time*), o que pode ser contraditório para um ambiente HPC, onde a curva de objetivo pode vir a ser de 100% do poder computacional.

Problemas de compatibilidade de aplicações podem inibir a adoção do *kernel tickless*, mas ele veio para ficar.

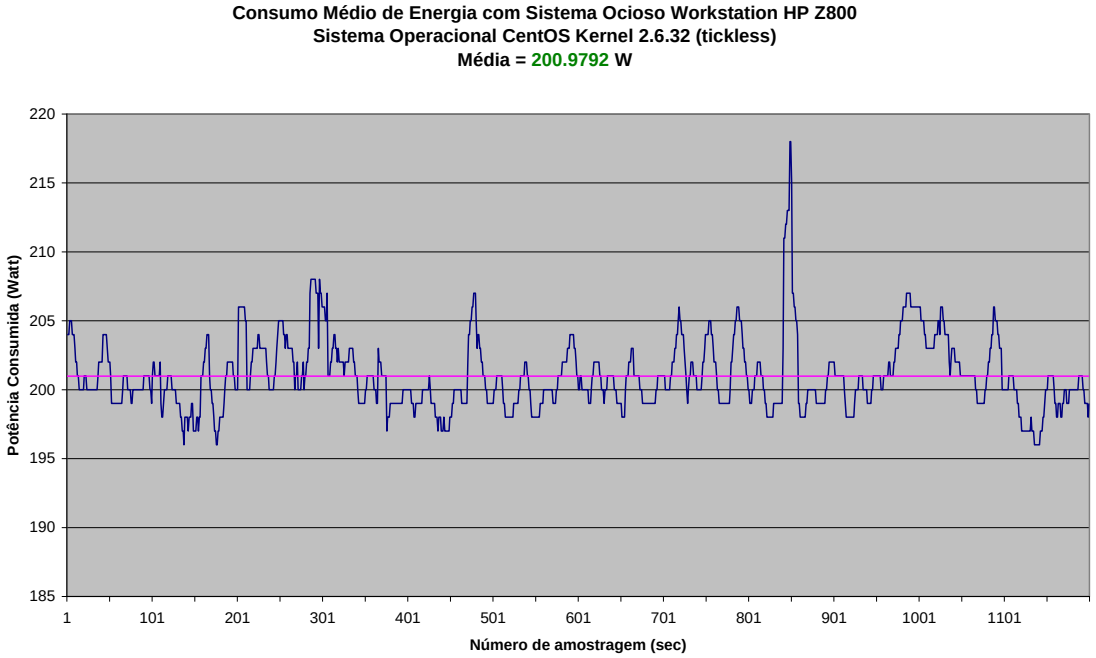

Figura A - 1 Consumo Médio de Energia com sistema operacional *tickless* em *idle time.*

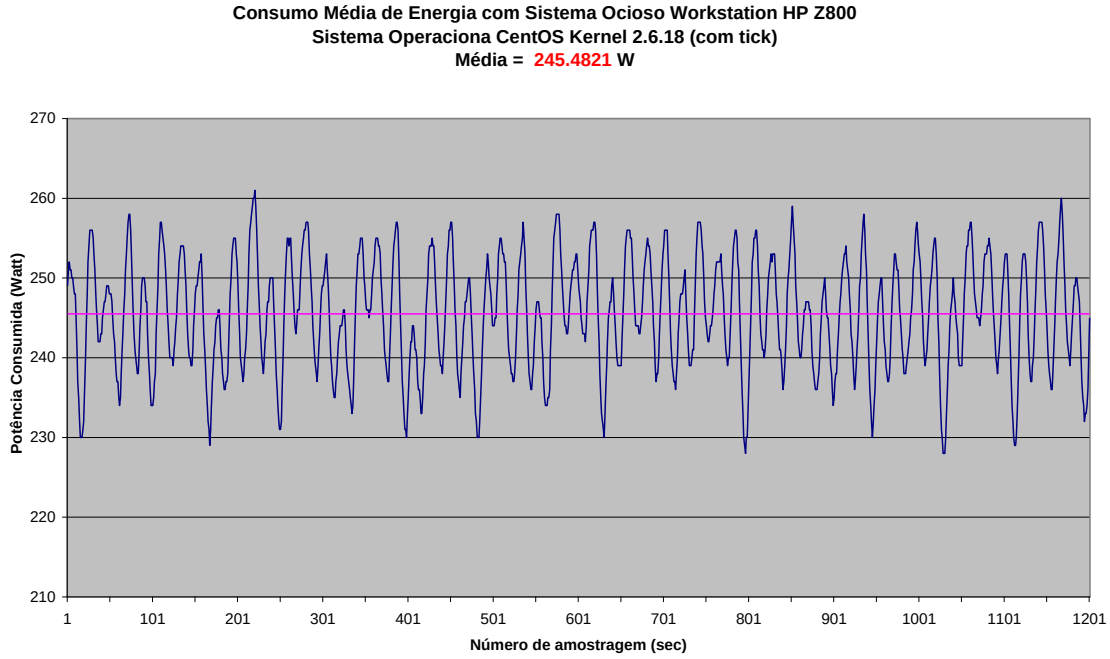

Figura A - 2 Consumo Médio de Energia com sistema operacional com *tick* em *idle time.*

HP Proliant 380

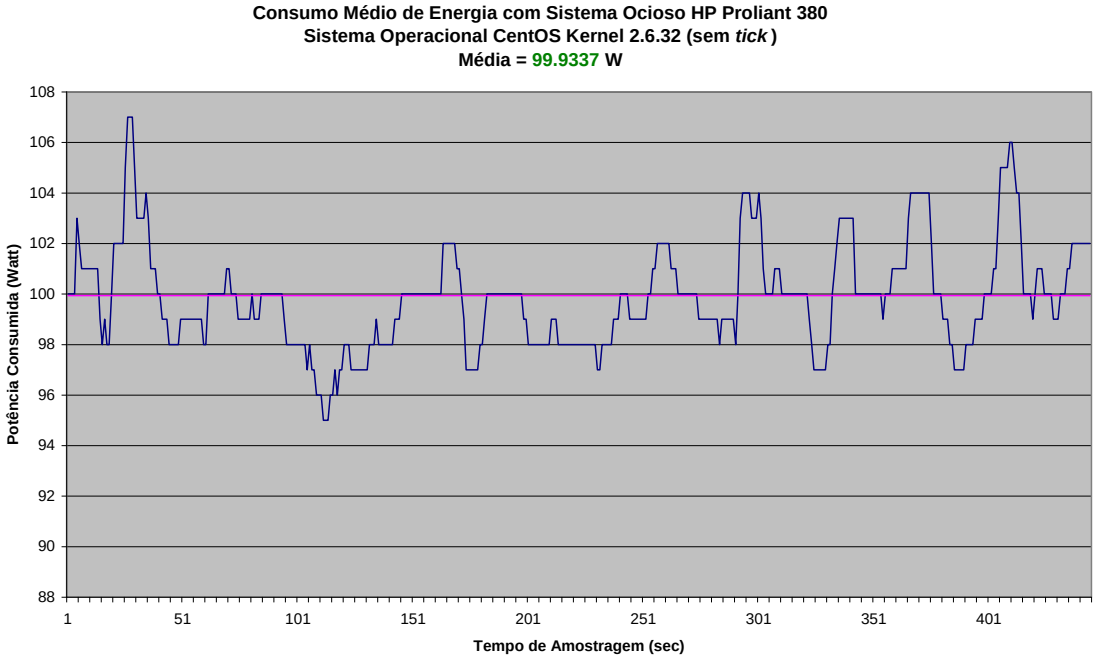

Figura A - 3 Consumo Médio de Energia com sistema operacional *tickless* em *idle time*

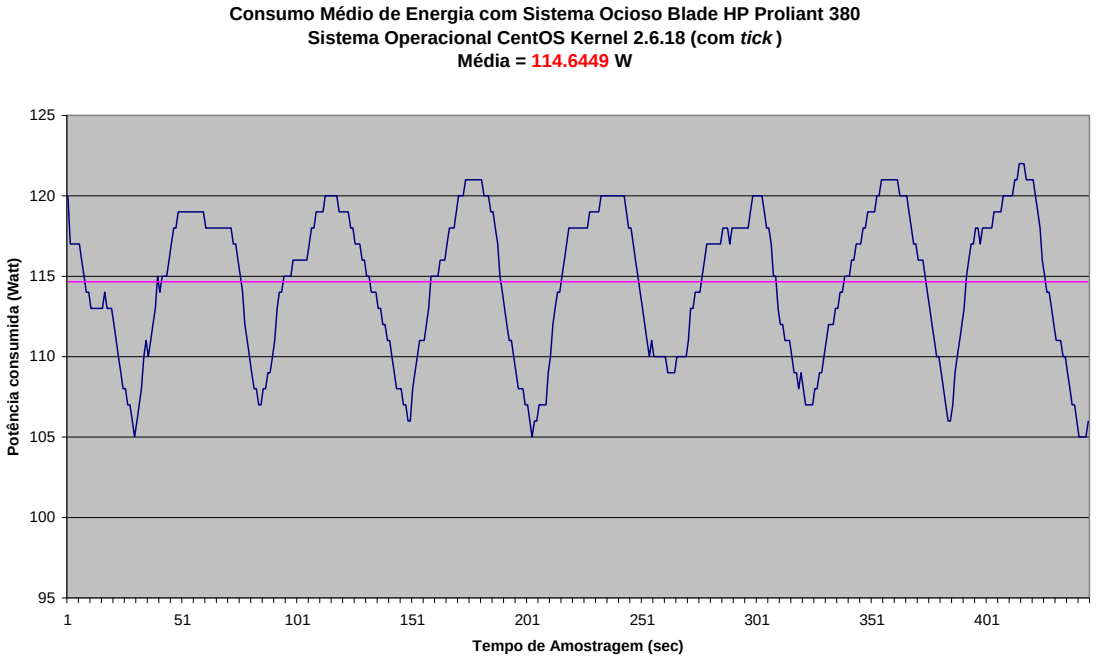

Figura A - 4 Consumo Médio de Energia com sistema operacional com *tick* em *idle time*.

ANEXO A – ANATOMIA DO ESCALONADOR CFS

Estrutura de Dados

As mais importantes estruturas de dados usadas no Linux são ***struct task_struct*** e ***struct rq*** e ***struct sched_class***. Os principais métodos do *kernel* trabalham sobre estas estruturas.

- **Estrutura *task_struct***

Cada processo no sistema é representado por uma *task_struct*. Como o tipo de dado *task_struct* deve ser capaz de capturar todas as informações de um processo, ele é relativamente longo: aproximadamente 1,7 KB. Quando um processo/*thread* é criado, o *kernel* aloca uma nova *task_struct.* O *kernel* então armazena a *task_struct* em uma lista circular encadeada denominada *task_list*. Existe uma conveniente macro *current* para se obter o corrente ou atual processo em execução de uma dada fila de execução associada a um específico processador. Outras macros, *next_task* e *prev_task*, permitem um processo para que se obtenha sua próxima tarefa e sua tarefa anterior, respectivamente. O escalonador do Linux usa muitos outros campos da *task_struc* para sua tarefa de escalonamento. Dentre outros, podemos citar: *state, state-prio, prio, normal_prio, rt_priority, sched_class, sched_entity, sched_rt_entity, policy*, etc.

- **Estrutura *rq***

Em tempo de boot, o sistema Linux cria filas de execução do tipo ***rq*** para cada processador disponível. Cada fila de execução captura os seguintes dados:

nr_running: número de tarefas executáveis
nr_switches: número de trocas
cfs: estrutura das filas de execução **CFS**
RT: estrutura das filas de execução em tempo-real (**RT**)
next-balance: *data e hora* para a próxima verificação de balanceamento de carga
curr: *ponteiros* apontando para a tarefa em execução corrente desta fila de execução
idle: *ponteiros* apontando para a tarefa ociosa corrente desta fila de execução

lock: *spinlock* da fila execução, *task_rq_lock()* e *task_rq_unlock()* podem ser usados para bloquear a fila de execução com uma específica tarefa em execução. É importante que os bloqueios da fila de execução ocorram na mesma ordem no caso do bloqueio de múltiplas filas.

- **Estrutura *sched_class*: *fair_sched_class* e *rt_sched_class***

A estrutura *sched_class* foi e é uma novidade inserida no *kernel* 2.6.23. A intenção é separar a lógica das políticas de escalonamento do arquivo principal *sched.c*. Isto torna o código mais modularizado. Alguns dos principais métodos definidos na *sched_class* são: *enqueue_task(), dequeue_tree(), yield_task(), check_preempt_curr(), pick_next_task(), set_curr_task(), task_tick(), task_new().*

Através do método *task_new(),* por exemplo, o núcleo do escalonador permite ao módulo de escalonamento uma oportunidade de gerenciar a inicialização de novas tarefas. O módulo de escalonamento *CFS* usa este método para o escalonamento de grupos, enquanto o módulo de escalonamento para tarefas em tempo real não o usa.

Escalonamento de Tarefas

Os principais métodos para o escalonamento de tarefas são: *schedule(), context_switch*() e *schedule_fork*().

- ***schedule()***

É o método mais importante do escalonador. Este método é responsável por pegar a próxima tarefa e trocar da tarefa atual para a próxima tarefa. Este método é executado quando:

- um processo voluntariamente cede CPU
- um processo aguarda que aconteça um sinal ou quer dormir
- ocorre a tarefa de temporização pelo método *scheduler_tick()*

Passos do método *schedule()*:

1 A preempção é desabilitada. Durante a execução da lógica de escalonamento, a preempção deve ser desabilitada. Isto porque obteremos um *spinlock* mais tarde.

2 Recupera a fila de execução associada ao processador corrente. Nós invocamos o método *smp_processor_id()* para obter a identificação da CPU corrente. Chama *cpu_rq()* para recuperar a fila de execução ***rq*** desta CPU. Então, chama *release_kernel_lock()* para liberar o bloqueio do *kernel* sobre a tarefa corrente e obter o bloqueio da fila de execução ***rq*** corrente.

3 Executa o método *pre_schedule()*. Lembremos que o método *pre_schedule()* é definido na estrutura *sched_class()*. Isto é um gancho para a política de escalonamento executar qualquer lógica antes de chamar o método *pick_next_task()*. Atualmente, apenas a classe de escalonamento RT de tempo real implementa este método. Isto se deve ao fato de que o Linux favorece as tarefas de tempo-real em detrimento das tarefas ditas normais. Implementar o método *pre_schedule()* na classe de escalonamento RT permite ao escalonador RT verificar se existem tarefas de tempo real a executar neste ponto. Se existir uma tarefa de tempo-real executável no sistema, o escalonador RT tornará esta tarefa como uma tarefa executável corrente.

4 Executa o método *pick_next_task()*. Este método é chamado associado a classe de escalonamento armazenada na *task_struct.sched_class* da tarefa corrente. Se a tarefa corrente é uma tarefa normal, *pick_next_task()* da classe de escalonamento ***CFS*** será executada, que seguira o algoritmo ***CFS*** (Escalonamento Completamente Equânime) para a próxima tarefa a ser obtida. Se a tarefa corrente é uma tarefa de tempo-real, *pick_next_task()* da classe de escalonamento RT será executada, em conformidade com o padrão POSIX de tempo real.

5 Verifica se a tarefa corrente é a mesma próxima tarefa a ser executada. Se for a mesma, não é preciso troca de contexto. Simplesmente libera o bloqueio da fila de execução e executa o método *post_schedule()*. Em caso contrário, a mudança de contexto é obrigatória para a troca da tarefa corrente para a próxima e distinta tarefa. Quando a troca de tarefas é realizada, o código chama *post_schedule()*.

6 *post_schedule()* é um gancho que permite que classe de escalonamento RT insira tarefas de tempo real no processador corrente.

7 Adquire bloqueio na tarefa corrente que deve ser a tarefa original ou a próxima tarefa e habilita a preempção. O código então verifica se é preciso re-escalonar novamente; se sim, voltamos ao passo correspondente e refazemos o reescalonamento.

- ***context_switch()***

A troca de contexto é a forma pela qual o sistema operacional, através do escalonador de processos, efetua a substituição ou troca propriamente do processo corrente que detém o controle da CPU por um seu sucessor, disponibilizando as instruções e dados pertinentes ao processo futuro a ser executado [65].

O método *context_switch()* é chamado a partir do *schedule()* para efetuar a substituição da tarefa corrente pela próxima tarefa. Este método faz o trabalho específico da máquina para o processo de troca de contexto, ou seja, comutação de memória, registradores e pilha.

Passos do método *context_switch()*:

1 Prepara a tarefa de troca de contexto chamando os métodos *prepare_lock_switch()* e *prepare_arch_switch()*.

2 Executa a troca de contexto de CPU. Isto colocará a CPU no estado preparado para a troca de contexto (CPU e MMU).

3 Inicia o processo de troca de contexto da memória virtual do processo chamando *switch_mm()*. Este passo usa diretamente o utilitário fornecido por diferentes processadores para a carga da tabela de páginas da próxima tarefa. Na arquitetura *x86*, usa o registrador *cr3* no método *load_cr3()* para completar a troca de memória. A maioria dos processos compartilha a mesma LDT (*Local Descriptor Table*). Se a LDT é necessária, nós a carregamos aqui a partir do novo contexto.

4 Troca os registradores do processo e a pilha do processo. Este passo envolve principalmente códigos em linguagem de máquina que são valores salvos do registrador *esp* corrente para o *prev* da estrutura da tarefa. Então, move o ponteiro da pilha da nova tarefa para o registrador *esp* processador corrente. Agora estamos em uma nova pilha do *kernel*. O código carrega o valor da nova pilha do *kernel* para o registrador da CPU. A macro *switch_to(prev, next, prev)* passa o *prev* de volta no registrador *eax*, que é o terceiro parâmetro. Noutras palavras, o parâmetro de entrada *prev* é passado para a macro *switch_to()* como último parâmetro de saída.

5 Então, requisitamos a inicialização da otimização da barreira de memória e depois chamamos *finish_task_switch()* para completar a troca de contexto.

- ***schedule_fork()***

O método *schedule_fork()* é chamado durante uma chamada de sistema *fork()/clone()*. Este método inicializa todos os campos relacionados ao escalonamento e definidos na estrutura *struct task_struct*.

Passos do método s*chedule_fork()*:

1. Recupera a CPU corrente no método *get_cpu()* desabilitando a preempção e obtendo o CPU *id* do sistema SMP.

2. Inicializa os valores para os campos da entidade de escalonamento da nova *task_struct*.

3. Configura o estado como *TASK_RUNNING*.

4. Se for obrigatório reconfigurar na chamada do *fork*, deve-se reconfigurar a nova *task_struct*. Caso contrário, configura-se a prioridade para a nova tarefa.

5. Configura-se a classe de escalonamento para a nova tarefa baseado em sua prioridade. Se a prioridade é uma prioridade de tempo real, a classe de escalonamento é a classe de escalonamento **RT**. Caso contrário, sua classe de escalonamento é configurada para a classe de escalonamento ***CFS***.

6. Configura a CPU *id* para o campo CPU da *task-struct*.

7. Coloca a nova tarefa na lista de processos chamando *plist_head_init()*.

8. Então, chama *put_cpu()* para habilitar a preempção.

Balanceador de Carga

O método *load_balance()* é chamado como parte do processo *schedule()*. Ele examina no domínio da CPU e tenta balancear as tarefas entre as CPUs mais ocupadas e as mais ociosas.

Passos do balanceamento de carga (*load_balance()*):

1. Configura todas as CPUs disponíveis na máscara de CPUs. Se existe uma CPU, a esta é associado um bit configurado como 1.

2. Usando a máscara acima, o código tenta encontrar o grupo mais ocupado. O método *find_busiest_group()* retorna o grupo mais ocupado dentro do *sched_domain* se existe um desbalanceamento. Se não existir um desbalanceamento, e o usuário optar por economia de energia, este retorna um grupo cujas CPUs podem ser colocadas no estado ocioso pelo rebalanceamento daquelas tarefas e em caso contrário, se o grupo existir. Também calcula a quantidade de carga ponderada que pode ser movida para restabelecer o equilíbrio de consumo de processamento entre as CPUs.

3. Verifica se pode ser encontrado um grupo mais ocupado. Se sim, o método continua. Senão, retorna à execução.

4. Ao se encontrar um grupo de processadores mais ocupado, busca-se a fila de execução mais ocupada deste grupo.

5. Obtém-se um duplo bloqueio na ordem: fila de execução corrente e fila mais ocupada.

6. Inicia o movimento da tarefa. O método *move_task()* tenta mover para o *max_load_move* a carga mais pesada da fila mais ocupada para esta fila de execução (*this_rq*), como parte da operação de balanceamento no domínio ***sd*** . Retorna 1 (um) se houver sucesso ou 0 (zero) em caso contrário.

7. Libera os bloqueios obtidos no passo 5 (liberando-os em ordem reversa ao da obtenção).

8. Verifica se a tarefa de movimentação no passo 6 foi bem sucedida. Se sim, o código zera o tempo de intervalo do balanceamento. Caso contrário, acorda a tarefa de migração para que o balanceamento de carga seja disparado mais tarde por esta tarefa – *load_balance()*.

9. Finalmente, o código atualiza a informação de CPU após o balanceamento.

Classe de Escalonamento *CFS*

O escalonador ***CFS*** foi projetado a partir de uma nova abordagem que nada tem a ver com qualquer matriz de prioridades, como o seus antecessores. O ***CFS*** tem sua escolha lógica baseada no valor *schedule_entity.vruntime* e é assim muito simples: ele tenta executar uma tarefa com o menor valor *p->se.vruntime* (isto é, a tarefa que tenha sido menos executada em tempo de processamento). O ***CFS*** sempre tenta dividir o tempo de CPU entre tarefas executáveis, tão próximo do "*hardware* multitarefa ideal e preciso", quanto possível. Um "*hardware* multitarefa ideal e preciso" é aquele teoricamente capaz de executar **N** processos concorrentes dando a cada um deles exatamente 1/N (um n-ésimo) de seu poder computacional de ***CPU*** disponível. Para sua estrutura de dados interna, o escalonador ***CFS*** usa uma árvore binária rubro-negra (*rb-tree*), que é uma árvore auto-balanceada. A complexidade de busca e atualização (inserção e exclusão) de um nó é da ordem de $O(\log n)$. O uso de árvore rubro-negra tem duas vantagens:

- Não tem artefatos de "troca de *arrays ativo ou expirado*" (pelos quais seus antecessores são afetados: ***O(n)***, ***O(1)***, ***RSDL/SD***).

- Confere um componente de auto-balanceamento da árvore rubro-negra para a redução da complexidade do escalonador.

O ***CFS*** mantém uma árvore rubro-negra ordenada no tempo, onde todas as tarefas executáveis são classificadas pela chave *vruntime* (existe uma subtração usando-se ***rq*** -> *cfs.min_vruntime* para levar em conta possíveis desvios). O ***CFS*** pega a tarefa contida no nó inferior "mais à esquerda" desta árvore e se fixa nele. Com o avançar do sistema, as tarefas executadas são colocadas na árvore cada vez mais e mais à direita lentamente para se certificar de dar uma chance para toda tarefa tornar-se "a tarefa mais à esquerda" e assim obter **CPU** dentro de um intervalo de tempo determinístico.

Os campos das árvores binárias rubro-negras podem ser visualizados e rastreados dinamicamente ou a qualquer tempo para efeito de análise, por exemplo, através dos dados obtidos via comando ou utilitário de depuração.

cat ***/proc/sched_debug*** > **arquivo.debug** (redirecionado para um arquivo de trabalho).

A implementação do método *pick_next_task_fair()* na classe de escalonamento ***CFS*** é muito simples graças a estrutura de dados representada em árvore rubro-negra.
Passos do método *pick_next_task_fair()*:

1. Verifica se existe qualquer tarefa executável na fila *cfs_rq*.
2. Chama *pick_next_entity()* que implicitamente chama *rb_entry()* para recuperar a próxima tarefa com o menor valor da chave *vruntime*.
3. O método *sets_next_entity()* executa a atualização dos valores da *sched_entity* do nó recuperado a partir do passo 2.
4. Chama o método *group_cfs_rq()* para retornar o grupo proprietário da próxima tarefa. O agrupamento é um aprimoramento que foi introduzido no *kernel* 2.6.24 para fins de equidade para usuários e grupos e não somente para tarefas. O escalonamento de grupos pode ser habilitado se *CONFIG_FAIR_GROUP_SCHED* é selecionado durante a geração do *kernel*. Se existir um grupo proprietário para esta tarefa, é efetuado o laço do passo 2.