PEDRO EUGÊNIO ROCHA

SISTEMAS DE ARMAZENAMENTO COMPARTILHADO COM QUALIDADE DE SERVIÇO E ALTO DESEMPENHO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Luis Carlos Erpen de Bona

CURITIBA

2011

PEDRO EUGÊNIO ROCHA

SISTEMAS DE ARMAZENAMENTO COMPARTILHADO COM QUALIDADE DE SERVIÇO E ALTO DESEMPENHO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Luis Carlos Erpen de Bona

CURITIBA

2011

PEDRO EUGÊNIO ROCHA

SISTEMAS DE ARMAZENAMENTO COMPARTILHADO COM QUALIDADE DE SERVIÇO E ALTO DESEMPENHO

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comisso formada pelos professores:

Orientador: Prof. Dr. Luis Carlos Erpen de Bona
Departamento de Informática, UFPR

Prof. Dr. Bruno Schulze
Departamento de Ciência da Computação, LNCC

Prof. Dr. Elias P. Duarte Jr.
Departamento de Informática, UFPR

Prof. Dr. Marcos Sunye
Departamento de Informática, UFPR

Curitiba, 07 de julho de 2011

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer a meus pais que me apoiaram das mais diversas formas, desde sempre, e tiveram um papel fundamental nesta caminhada. Obrigado, vocês são ótimos.

Gostaria também de agradecer a minha namorada, Gaby, que me acompanhou durante toda a jornada, dando-me apoio em momentos difíceis, comemorando momentos alegres, aturando minhas chatices, respeitando e entendendo o tempo dedicado a esta dissertação. Obrigado pelo companheirismo e carinho.

Não poderia deixar de agradecer a todos os amigos, parceiros de churrasco e de rodas de poker, companheiros de boteco, colegas de faculdade e de trabalho pelos conselhos, diversão e amizade. Agradeço especialmente aos colegas de trabalho do TRE-PR, por todo apoio nesta reta final.

Gostaria também de registrar meus sinceros agradecimentos ao orientador deste trabalho, prof. Bona. Obrigado por seu tempo desde a época da monografia, pelas revisões, pelos rabiscos no texto, por sua sabedoria. Gostaria ainda de agradecer aos membros da banca, especialmente ao prof. Elias, pela revisão minuciosa que em muito contribuiu para a melhoria deste trabalho. Agradeço também a todos os professores do C3SL, do departamento de informática e da universidade como um todo, pela instrução.

Finalmente, gostaria de agradecer a todos os brasileiros que indiretamente financiaram minha formação. Comprometo-me a retribuir o investimento no que estiver a meu alcance, trabalhando para tornar nosso país um lugar mais próspero e justo.

He who receives an idea from me, receives instruction himself without lessening mine; as he who lights his taper at mine, receives light without darkening me. Inventions then cannot, in nature, be a subject of property.

Thomas Jefferson

SUMÁRIO

LISTA DE FIGURAS v

RESUMO viii

ABSTRACT ix

1 INTRODUÇÃO 1

2 ESTRUTURAS DE ARMAZENAMENTO SECUNDÁRIO 4

2.1 O Disco Magnético 5

2.2 Comunicação com o Disco 7

2.3 Escalonamento de Disco 10

2.3.1 FCFS 10

2.3.2 SSTF 11

2.3.3 SCAN 11

2.3.4 C-SCAN 12

2.3.5 LOOK e C-LOOK 12

2.4 Acesso a Disco no Kernel Linux 13

2.4.1 Noop 15

2.4.2 Deadline 15

2.4.3 Anticipatory 16

2.4.4 CFQ 17

3 QUALIDADE DE SERVIÇO 19

3.1 Algoritmo do Balde Furado 21

3.2 Algoritmo do Balde de Símbolos 22

3.3 Algoritmos de Enfileiramento Justo 23

4 TRABALHOS RELACIONADOS **27**

4.1 Budget Fair Queueing 29

4.2 pClock 30

4.3 Comparação entre os Algoritmos 31

5 HIGH-THROUGHPUT TOKEN BUCKET SCHEDULER **32**

5.1 Visão Geral 33

5.2 O Algoritmo 35

5.3 Parâmetros 38

6 RESULTADOS EXPERIMENTAIS **40**

6.1 Comparação com pClock 40

6.2 Impacto do Atributo B_{max} na Latência 42

6.3 Utilização de Rajadas 44

6.4 Largura de Banda Acumulada 46

7 CONCLUSÃO **50**

BIBLIOGRAFIA **57**

LISTA DE FIGURAS

2.1 Estrutura interna de um disco. 6
2.2 Camada de bloco do Kernel Linux. 15

3.1 Ordem de envio de pacotes e atendimento pela disciplina GPS. 25
3.2 Ordem de atendimento utilizando os algoritmos WFQ e WF^2Q. 26

5.1 Modelagem do sistema. 33
5.2 Algoritmo principal do HTBS. 36
5.3 Funções que controlam a atribuição de *tags* e *tokens*. 37

6.1 Largura de banda para o algoritmo HTBS. 41
6.2 Largura de banda para o algoritmo pClock. 42
6.3 Latência utilizando o algoritmo HTBS com $B_{max} = 1$. 43
6.4 Latência utilizando o algoritmo HTBS com $B_{max} = 20$. 44
6.5 Rajadas utilizando o algoritmo pClock. 45
6.6 Rajadas utilizando o algoritmo BFQ. 46
6.7 Rajadas utilizando o algoritmo HTBS. 47
6.8 Largura de banda acumulada para os algoritmos pClock, BFQ e HTBS. . . 49

GLOSSÁRIO DE SIGLAS

AS	*Anticipatory Scheduler*
ATA	*Advanced Technology Attachment*
BFQ	*Budget Fair Queueing*
BWF^{2}Q	*Budget WF^{2}Q+*
CFQ	*Complete Fairness Queueing*
CPU	*Central Processing Unit*
DiffServ	*Differentiated Services*
DMA	*Direct Memory Access*
EDF	*Earliest Deadline First*
EEPROM	*Electrically Erasable Programmable Read-Only Memory*
EIDE	*Enhanced Integrated Drive Electronics*
EYFQ	*Extended Yet Another Fair Queueing*
FC	*Fiber Channel*
FCFS	*First Come First Served*
FD-EDF	*Feasible Deadline EDF*
FD-SCAN	*Feasible Deadline SCAN*
FIO	*Flexible I/O Tester*
GPS	*General Processor Sharing*
HDD	*Hard Disk Drive*
HTB	*Hierarchical Token Bucket*
HTBS	*High-throughput Token Bucket Scheduler*
IETF	*Internet Engineering Task Force*
IntServ	*Integrated Services*
IOMMU	*Input/Ouput Memory Management Unit*
IOPS	I/O por segundo
ISP	*Internet Service Provider*
LVM	*Logical Volume Manager*
MMU	*Memory Management Unit*
MPLS	*Multi Protocol Label Switching*
NCQ	*Native Command Queueing*
Noop	*No Operation*
OSI	*Open Systems Interconnection*
PGPS	*Packet General Processor Sharing*
PSCAN	*Priority SCAN*
QoS	*Quality of Service*
RAID	*Redundant Array of Independent Disks*

RAM	*Random Access Memory*
RPM	Rotações por Minuto
RSVP	*Resource Reservation Protocol*
SATA	*Serial ATA*
SCSI	*Small Computer System Interface*
SFQ	*Stochastic Fair Queueing*
SGBD	Sistemas de Gerenciamento de Banco de Dados
SLA	*Service Level Agreement*
SSTF	*Shortest Seek Time First*
TOS	*Type of Service*
USB	*Universal Serial Bus*
VFS	*Virtual Filesystem*
WFQ	*Weighted Fair Queueing*
WF^2Q	*Worst-case Fair Weighted Fair Queueing*
YFQ	*Yet Another Fair Queueing*

RESUMO

É desejável que servidores de armazenamento compartilhado possuam mecanismos que forneçam isolamento de performance entre aplicações concorrentes. Em sistemas com isolamento de performance, cada aplicação recebe a quantidade de recurso determinada por suas garantias de QoS, especificadas em SLA's, independentemente da carga e dos padrões de utilização de outras aplicações. Uma abordagem muito utilizada atualmente é fornecer garantias de QoS e isolamento de performance através do algoritmo de escalonamento de disco. Contudo, para que isso seja possível sem que o desempenho do disco seja degradado, também é necessário que o algoritmo previna a ocorrência de *deceptive idleness*. *Deceptive idleness* é a situação onde uma aplicação é considerada ociosa pelo escalonador de disco, no instante imediatamente posterior ao término do atendimento de uma requisição síncrona, mas anterior ao envio da próxima. Assim, caso uma aplicação seja erroneamente considerada ociosa, um escalonador conservativo atenderia outra aplicação, perdendo a localidade no acesso e diminuindo o desempenho geral do dispositivo. Este trabalho propõe um novo algoritmo de escalonamento de disco não-conservativo, o *High-throughput Token Bucket Scheduler* (HTBS), que utiliza conceitos retirados de outros dois algoritmos de escalonamento de disco: BFQ e pClock. O HTBS incorpora a ideia de prevenção de *deceptive idleness* adotada pelo BFQ, mesclada à política de atribuição de *tags* e atendimento de requisições do algoritmo pClock. Mostramos, através de experimentos realizados com a implementação do HTBS para o Kernel Linux, que o HTBS pode fornecer garantias de QoS para aplicações com requisitos variados mesmo na presença de requisições síncronas, bem como alto desempenho.

ABSTRACT

Shared storage systems must provide performance isolation among concurrent applications. In such systems, each application receives a specified share, according to its QoS guarantees defined in SLA's, regardless the behavior and load of other applications. A very common approach in nowadays storage systems is to provide QoS guarantees and performance isolation through the disk scheduler algorithm. However, in order to provide these guarantees without decreasing system performance, it is also required that the scheduler prevents *deceptive idleness*. *Deceptive idleness* is a situation where an application is deemed idle by the disk scheduler, in the moment right after a synchronous request completion but prior the issue of the next one. Therefore, if an application was mistakenly deemed idle, a work-conserving disk scheduler would switch service to another application, losing locality and decreasing disk overall performance. This work proposes a new *non-work-conserving* disk scheduling algorithm, called High-throughput Token Bucket Scheduler (HTBS), which is based on concepts of two former disk schedulers: BFQ and pClock. HTBS merges the deceptive idleness prevention mechanism from BFQ and the timestamp attribution policy and request serving order from pClock. We show, through experiments performed with the HTBS implementation for Linux Kernel, that HTBS can provide both high throughput and QoS guarantees to applications with different attributes even when using synchronous requests.

CAPÍTULO 1

INTRODUÇÃO

A centralização da capacidade de armazenamento em servidores dedicados é uma abordagem cada vez mais utilizada no gerenciamento de dados organizacionais. Dentre os diversos benefícios trazidos por essa centralização estão a redução de custos e da complexidade de manutenção, a utilização otimizada dos recursos de *hardware*, a simplificação no gerenciamento e em políticas de backup, além da flexibilidade para a alocação de espaço de armazenamento [1]. Além disso, a crescente utilização da computação em nuvens aumentou a demanda por novos serviços de armazenamento que pudessem ser compartilhados entre diferentes usuários [2, 3, 4].

Para que o recurso de armazenamento seja compartilhado, é necessário que o sistema forneça, para cada usuário, a abstração de um disco dedicado, também chamado de *disco virtual*. Além da capacidade de armazenamento (geralmente medida em *gigabytes*, *terabytes* ou *petabytes*), estes discos virtuais também devem possuir garantias de QoS, que, definidas em documentos chamados SLA's (*Service Level Agreements*) [4], especificam e mensuram as garantias de desempenho no acesso ao disco. Tais garantias são baseadas em atributos de QoS, como largura de banda, latência e rajadas.

Uma das possíveis soluções para o gerenciamento de discos virtuais é o mapeamento estático, isto é, a alocação de um disco físico para cada usuário. Além de não ser flexível, esta estratégia demanda uma quantidade maior de recursos de *hardware* do que o realmente necessário [5]. Uma segunda estratégia é a utilização de mecanismos de *isolamento de performance* [6], compartilhando um mesmo recurso entre diferentes aplicações. Em sistemas com isolamento de performance, cada aplicação recebe a quantidade de recurso especificada, independentemente da carga do sistema.

Este trabalho foca nos sistemas de armazenamento baseados em discos rígidos, pois, principalmente em conjunto com camadas intermediárias de *software* como RAID (*Re-*

dundant Array of Independent Disks) e LVM (*Logical Volume Manager*) [7], são os mais utilizados atualmente [8, 9]. Entretanto, fornecer garantias de QoS no acesso a disco é uma tarefa difícil, pois o tempo necessário para atender requisições de leitura ou escrita é altamente variável, dependendo de fatores como *seek time*, latência de rotação, posição dos dados na superfície do disco e caches [10, 11]. A ordem em que as requisições são executadas também tem grande impacto no desempenho do dispositivo. Ainda assim, podem existir diversas aplicações com características e requisitos de QoS variados competindo pela utilização do disco.

O impacto destes fatores sobre o desempenho do disco pode ser minimizado caso as requisições de acesso sejam executadas em ordem adequada. Algoritmos de escalonamento de disco têm a função de reorganizar estas requisições, de forma que a performance do dispositivo seja maximizada, ou visando implementar determinada política de acesso. Seja qual for o propósito do algoritmo, aplicações que utilizam requisições síncronas podem prejudicar o resultado esperado do escalonamento [12]. Nestas aplicações, a criação de uma nova requisição depende do atendimento da anterior; desta forma, apenas uma requisição de cada aplicação síncrona é mantida pendente pelo escalonador em determinado instante.

Como há relação de dependência entre requisições síncronas de uma mesma aplicação, existe um curto intervalo de tempo entre o término do atendimento de uma requisição e a criação da próxima, durante o qual a aplicação é dita enganosamente ociosa, ou *deceptively idle* [13]. Neste intervalo, como ainda não há novas requisições pendentes da mesma aplicação, um algoritmo de escalonamento conservativo passaria a servir uma próxima aplicação, e assim sucessivamente. Esta constante troca da aplicação recebendo serviço, também conhecida como *deceptive idleness*, prejudica o desempenho do disco por não se beneficiar da localidade de requisições de uma mesma aplicação [12, 13, 14].

Neste trabalho é proposto um novo algoritmo de escalonamento de disco, chamado HTBS (*High-throughput Token Bucket Scheduler*), que utiliza conceitos baseados principalmente em dois algoritmos propostos anteriormente: BFQ (*Budget Fair Queueing*) [12] e pClock [5]. O HTBS tem o objetivo de fornecer garantias de QoS, causando o menor impacto possível no desempenho do disco. O algoritmo pClock é utilizado pois provê bom

isolamento de performance (como mostrado em [5]), além da possibilidade de ajustar os parâmetros largura de banda, rajadas e latência de forma independente. Não obstante, mostramos através de experimentos que o algoritmo pClock falha em fornecer as garantias na presença de requisições síncronas.

O algoritmo proposto incorpora os mecanismos para o tratamento de requisições síncronas e prevenção de ocorrência de *deceptive idleness* presentes no algoritmo BFQ. Tais mecanismos, mesmo que já utilizados em algoritmos anteriores (como *Anticipatory* [14] e CFQ [15], por exemplo), são utilizados em conjunto com políticas de isolamento de performance no BFQ. Apesar disso, o BFQ não permite a configuração dos atributos largura de banda e latência de forma individual, ou mesmo o suporte a rajadas.

Para a realização de experimentos, o algoritmo HTBS foi implementado como um módulo para o Kernel Linux [16], implementando sua interface padrão para algoritmos de escalonamento de disco. Através dos testes realizados — utilizando a ferramenta de *benchmark* de disco *fio* [17] — é mostrado que o algoritmo HTBS aumenta o desempenho geral do disco na presença de aplicações síncronas e sequenciais, se comparado ao algoritmo pClock, provendo isolamento de performance e sem degradar de maneira significativa a latência das requisições. Também é mostrado que o HTBS fornece desempenho próximo ao BFQ que possui mecanismos para prevenção de *deceptive idleness*, mas não possibilita a configuração dos atributos largura de banda e latência de forma explícita e individual. Por último, através de experimentos também é mostrado que o HTBS possibilita a utilização de rajadas de requisições.

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 contém uma visão geral do funcionamento das estruturas de armazenamento secundário, assim como uma descrição dos algoritmos clássicos de escalonamento de disco e das estruturas de acesso a disco do Kernel Linux. No Capítulo 3 são discutidos alguns tópicos de Qualidade de Serviço, bem como algoritmos utilizados nesta área. No Capítulo 4 são descritos os trabalhos relacionados. O funcionamento do algoritmo do HTBS, seus parâmetros e outras definições são detalhadas no Capítulo 5, enquanto o Capítulo 6 apresenta os resultados obtidos através de experimentos. Finalmente, o Capítulo 7 conclui este trabalho.

CAPÍTULO 2

ESTRUTURAS DE ARMAZENAMENTO SECUNDÁRIO

A memória principal, geralmente implementada em módulos de memória RAM (*Random Access Memory*), é a única área de armazenamento de propósito geral que pode ser acessada diretamente pela CPU. Apesar da alta velocidade de acesso, essas memórias são voláteis, ou seja, os dados armazenados são perdidos caso a alimentação de energia seja interrompida. Ainda, muitas vezes a memória principal não possui espaço suficiente para armazenar a demanda de dados e programas do sistema. Por estes motivos, a maior parte dos computadores possui uma extensão da memória principal, chamada memória secundária, com o objetivo de armazenar grandes quantidades de dados de forma não-volátil, ou permanente [18].

Todos os dispositivos de armazenamento secundário são denominados dispositivos de bloco, ou seja, as operações — leituras e escritas — são realizadas sobre quantidades fixas de dados, chamados blocos ou setores. Os dois tipos de dispositivos de armazenamento secundário mais utilizados atualmente são: memórias *flash* e discos magnéticos.

A memória *flash* é um tipo especial de EEPROM (*Electrically Erasable Programmable Read-Only Memory*) muito utilizada em cartões de memória, telefones celulares, câmeras digitais e até mesmo em *notebooks* [19]. Por serem de natureza elétrica, essas memórias são de rápido acesso e têm maior tolerância a pressão, temperatura e choques mecânicos que os discos magnéticos.

Os discos armazenam dados em superfícies magnéticas, executando operações através de cabeças de leitura móveis que se deslocam sobre estas superfícies. Os discos, em geral, têm maior capacidade de transferência que memórias flash, mas possuem um *overhead* intrínseco causado pelo tempo de posicionamento das cabeças de leitura. Apesar deste *overhead*, os discos são os dispositivos de armazenamento secundário mais utilizados nos sistemas computacionais atuais, pelo seu baixo custo e grande capacidade de armazena-

mento.

Há ainda os dispositivos de armazenamento removíveis de baixo custo em relação à capacidade de armazenamento, também conhecidos como dispositivos de armazenamento terciário. CD-ROM's, DVD's, disquetes, fitas, *pendrives* e cartões de memória são exemplos deste tipo de dispositivo. O acesso a esses dispositivos é, em geral, mais lento que o acesso a dispositivos de armazenamento secundário, apesar de também serem classificados como dispositivos de bloco. Como o foco deste trabalho são sistemas com grande capacidade de armazenamento e rápido acesso, é assumida a utilização exclusiva de discos magnéticos.

O restante deste capítulo está organizado da seguinte forma. A Seção 2.1 descreve a arquitetura e funcionamento interno dos discos magnéticos, enquanto a Seção 2.2 explica como ocorre a comunicação entre o computador e o dispositivo. Na Seção 2.3 são descritos os algoritmos clássicos de escalonamento de disco, e, na Seção 2.4, são discutidas algumas especificidades da camada de dispositivos de bloco no Kernel Linux.

2.1 O Disco Magnético

O disco magnético, também chamado de disco rígido, HDD (*Hard Disk Drive*) ou simplesmente disco, é um dispositivo não-volátil com capacidade de armazenamento de grandes quantidades de dados. Um disco é composto por lâminas metálicas circulares extremamente rígidas, com o formato semelhante a CD-ROM's, denominadas *platters*. Por sua vez, os *platters* são revestidos por uma subtância magnética, permitindo que dados sejam armazenados em suas superfícies. Hoje em dia, apenas os discos de grande capacidade utilizam mais de 4 *platters*; a maioria dos discos de capacidade média utiliza 2 ou 3. O diâmetro dos *platters* determina o tamanho físico do disco, podendo variar de 1.8 a 5.25 polegadas, embora atualmente os discos de 3.5 polegadas sejam mais comuns [20].

A Figura 2.1 ilustra a estrutura interna de um disco. A superfície de um *platter* é logicamente dividida em faixas circulares concêntricas, chamadas *tracks* ou trilhas. Cada *platter* possui, geralmente, mais de 3000 trilhas, numeradas em ordem crescente da parte externa do disco ao centro. Para facilitar ainda mais a localização de dados no disco,

cada trilha é subdividida em setores ou *sectors* — pequenos trechos de uma trilha onde os dados são armazenados. Cada trilha pode possuir em torno de 900 setores, sendo cada setor composto por *header*, área de dados (geralmente 512 bytes) e *trailer*. *Header* e *trailer* armazenam informações como o número do setor e códigos de detecção e correção de erros, utilizados somente pela controladora *on-board* do disco.

Como mostrado à direita da Figura 2.1, para cada superfície de cada *platter* existe uma cabeça de leitura, responsável por realizar operações sobre as informações armazenadas. Todas as cabeças de leitura estão ligadas a uma mesma peça, chamada braço de leitura, impossibilitando o movimento individual das cabeças. Desta forma, em determinado instante, todas as cabeças de leitura estão posicionadas sobre a mesma trilha, em seu respectivo *platter*. O conjunto de trilhas sob as cabeças de leitura em determinada posição do braço é chamado cilindro.

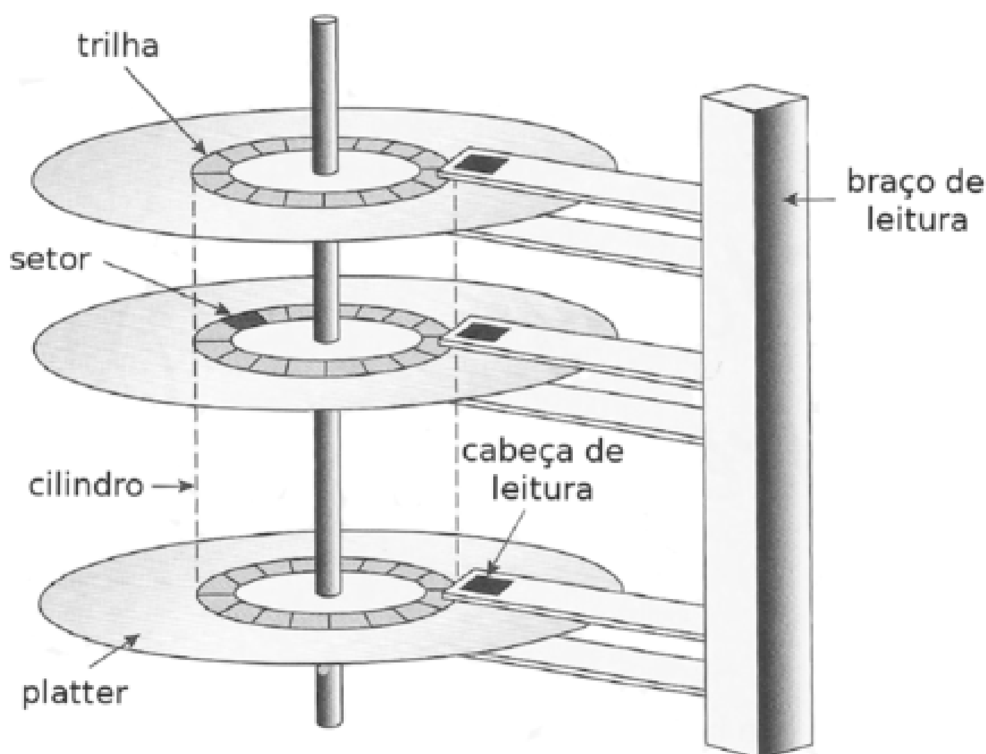

Figura 2.1: Estrutura interna de um disco [18].

O disco possui também um motor responsável por rotacionar os *platters*. Enquanto os discos mais antigos utilizavam motores de 3.600 ou 5.600 rpm (Rotações por Minuto), os

discos atuais possuem motores de 7.200, 10.000, 15.000 rpm ou mais [20]. A velocidade do motor de rotação é um dos fatores determinantes no desempenho de um disco, pois limita o tempo de posicionamento do setor sob a cabeça de leitura, uma vez que o braço de leitura esteja na posição correta.

Outras características relevantes ao desempenho dos disco são a *taxa de transferência de dados* e o *tempo de posicionamento*. A taxa de transferência de dados é a taxa em que os dados trafegam entre o disco e o computador. Uma taxa comum atualmente em discos de 7.200 rpm é 70 megabytes por segundo, dependendo fortemente do tipo do barramento e do posicionamento físico dos dados acessados no interior do disco. Por exemplo, dados gravados em trilhas externas tendem a trafegar em maior velocidade do que os gravados em trilhas internas (mais próximas ao centro do *platter*), pois essas possuem maior número de setores — cerca de 40% [20]. Já o tempo de posicionamento, também chamado de *tempo de acesso aleatório*, é calculado através de dois parâmetros: o tempo para posicionar o braço de leitura no cilindro desejado, chamado *seek time*, e o tempo para rotacionar o disco, de forma a posicionar o setor desejado sob a cabeça de leitura, chamado de *rotational latency* ou latência rotacional. A minimização do *seek time* é um dos principais desafios no escalonamento de requisições de disco.

2.2 Comunicação com o Disco

Os computadores atuais consistem, basicamente, em uma ou mais unidades de processamento (CPU) e diversas controladoras de dispositivos — também conhecidas como controladoras de I/O ou controladoras de E/S. A CPU e as controladoras de I/O estão conectados através de um barramento que provê acesso a uma memória compartilhada, ou memória principal. Cada controladora gerencia um tipo específico de dispositivo, como por exemplo, dispositivos de som, discos magnéticos, dispositivos de vídeo e a própria memória principal. Em alguns casos, diversos dispositivos podem ser conectados a uma mesma controladora, como em controladoras SCSI e USB, por exemplo [18].

As controladoras de I/O possuem registradores e buffers locais, que armazenam instruções e dados, respectivamente. A comunicação entre a CPU e as controladoras pode

ocorrer de duas formas: através de *memory-mapped I/O* ou de *port-mapped I/O*. Utilizando *memory-mapped I/O*, ou I/O mapeada em memória, faixas de endereçamento de memória são alocadas para os registradores e buffers das controladoras de I/O. Quaisquer operações — escritas ou leituras — nestas faixas de endereçamento da memória principal, serão executadas diretamente na memória da controladora. Contudo, para possibilitar o mapeamento de I/O em memória, todas as controladoras devem monitorar o barramento de endereçamento da CPU, atendendo requisições a endereços pertencentes ao espaço a que foram mapeados. No caso de *port-mapped I/O*, a comunicação é realizada através de instruções específicas da CPU, como as instruções *IN* e *OUT* nos microprocessadores Intel, por exemplo. Essas instruções leem e escrevem *bytes* nos registradores das controladoras de I/O. Utilizando esta abordagem, os dispositivos de I/O possuem um espaço de endereçamento independente da memória principal, especialmente vantajoso para CPU's com capacidade de endereçamento limitada [18]. Os endereços deste novo espaço de endereçamento são chamados de *I/O ports* ou portas de I/O.

Apesar de serem utilizadas por grande parte dos dispositivos de I/O, essas duas formas de comunicação (*memory-mapped* e *port-maped I/O*) são dispendiosas à CPU quando o objetivo é transferir grandes quantidades de dados — em acessos a disco, por exemplo. Para evitar desperdício de ciclos de CPU com esse tipo de operação, muitos computadores possuem uma controladora especial, chamada controladora DMA (*Direct Memory Access*) [21]. A controladora DMA opera diretamente o barramento de memória, realizando transferências sem o auxílio da CPU. Requisições ao disco são tipicamente realizadas através desta controladora. Para realizar uma operação DMA, a CPU cria um bloco de comando DMA em memória, contendo ponteiros para a origem e o destino dos dados, assim como o número de bytes a serem transferidos. O endereço deste bloco de comando é então escrito na controladora DMA que inicia a operação, deixando a CPU livre para a realização de outras tarefas. Ao final da transferência, a controladora DMA interrompe a CPU, avisando-a que os dados estão disponíveis.

Alguns computadores mais novos, particularmente os utilizados para virtualização, possuem ainda outra controladora, chamada IOMMU ou *Input/Output Memory Mana-*

gement Unit [22]. A IOMMU conecta dispositivos de I/O (cujos dados são transferidos através de DMA) à memória principal, realizando o mapeamento entre os acessos. Além disso, a IOMMU implementa mecanismos de proteção, de forma que os dispositivos só possam acessar as páginas de memória alocadas, de acordo com a tabela de páginas de I/O. Este processo é análogo ao que acontece na tradução de endereço virtual em endereço físico e proteção de memória, na MMU (*Memory Management Unit*).

Devido à complexidade de alguns dispositivos, certos periféricos possuem uma controladora embutida, como é o caso dos discos magnéticos, além da controladora de I/O do computador. A controladora embutida dos discos, também chamada de *disk controller*, é responsável, basicamente, por ler e escrever dados na sua superfície magnética, controlar a velocidade de rotação, movimentar o braço de leitura e comunicar-se com a controladora de I/O do computador, muitas vezes chamada de *host controller* ou controladora *onboard*. Existem diversos tipos de barramento que viabilizam a comunicação entre *disk* e *host controller*, como EIDE (*Enhanced Integrated Drive Electronics*), ATA (*Advanced Technology Attachment*), USB (*Universal Serial Bus*), FC (*Fiber Channel*) e SCSI (*Small Computer System Interface*). Os *disk controllers* usualmente possuem memória cache embutida, de forma que a transferência de dados entre o disco e a CPU ocorra primeiramente da superfície magnética do disco para a sua memória cache, e finalmente da memória cache à controladora de I/O [18]. Em discos mais novos, para cada setor acessado, toda a sua trilha é armazenada em memória cache no disco, aumentando o desempenho de acessos subsequentes a esta mesma trilha [23]. Esta memória cache, presente no próprio dispositivo, também é chamada de memória cache *onboard*.

Dispositivos periféricos com finalidades semelhantes podem utilizar diferentes controladoras de I/O, os quais possuem formas diferenciadas de acesso. Discos magnéticos podem utilizar controladoras SATA, SCSI ou IDE, por exemplo. Visando contornar esta variação, os sistemas operacionais agrupam os dispositivos segundo certas características, e definem interfaces comuns de acesso. A implementação de uma interface para determinada controladora é chamado de *device driver*, ou simplesmente *driver*. Esta abordagem abstrai as diferenças e peculiaridades do hardware das aplicações, e beneficia os fabricantes de

hardware, que apenas desenvolvem *drivers* que operem corretamente seus dispositivos e implementem essas interfaces.

2.3 Escalonamento de Disco

Todo acesso a disco é realizado através de *requisições*. Requisições podem ser atendidas instantaneamente, se o dispositivo estiver ocioso, ou, caso o dispositivo esteja ocupado, podem ser armazenadas em uma fila de requisições pendentes. É comum haver diversas requisições pendentes nesta fila, em sistemas onde existam diversos processos executando concorrentemente. Visando diminuir a influência do *seek time* e da latência de rotação no desempenho dos discos, o sistema operacional normalmente reordena essas requisições, de forma a maximizar a largura de banda (razão entre o número total de bytes transferidos e o tempo gasto pelas requisições) e diminuir a latência no acesso. São apresentados nas próximas Seções alguns algoritmos clássicos de escalonamento de disco, utilizados como base para grande parte dos escalonadores de disco dos sistemas operacionais atuais.

2.3.1 FCFS

O FCFS (*Fist Come, First Served*) é o mais simples dos algoritmos de escalonamento de disco. No FCFS, as requisições são atendidas exatamente na ordem em que foram criadas, não sendo realizadas quaisquer otimizações. Apesar de não proporcionar grande desempenho, por desconsiderar o *seek time* entre as requisições, o FCFS utiliza o disco de forma relativamente justa entre os processos. Este algoritmo gera muito pouca variação na latência entre requisições, prevenindo também a ocorrência de inanição[1] (também chamada de *starvation*) no sistema.

Exemplificando, considere que inicialmente a cabeça de leitura esteja posicionada sobre o cilindro 98, e as requisições pendentes na fila desejem acessar dados nos seguintes cilindros: 32, 16, 112, 87, 184, 105, 21 e 140. Utilizando o algoritmo FCFS, o braço de leitura deslocar-se-á do cilindro 98 (posição inicial neste exemplo) para o 32, atendendo a primeira requisição. Logo após, irá movimentar-se para o cilindro 16, servindo a próxima

[1]No caso dos discos, inanição ocorre quando uma requisição nunca é escalonada para atendimento.

requisição na fila, e desta mesma forma para os cilindros 112, 87, 184, 105, 21 e finalmente para o 140. O deslocamento total do braço de leitura necessário para atender este conjunto de requisições utilizando o FCFS, é de 582 cilindros.

2.3.2 SSTF

O tempo desperdiçado movimentando o braço de leitura (*seek time*) pode ser diminuído significativamente, se, antes de movê-lo para um cilindro distante, as requisições próximas a sua posição atual forem atendidas. Essa é a ideia básica do algoritmo SSTF [24] (*Shortest Seek Time First*), que prioriza as requisições mais próximas ao braço de leitura, visando diminuir o número de cilindros percorridos, e consequentemente o *seek time*.

Voltando ao exemplo anterior, inicialmente o braço de leitura encontra-se posicionado sobre o cilindro 98. Ao contrário do algoritmo anterior, que deslocou o braço de leitura até o cilindro 32, o SSTF atenderá primeiramente a requisição mais próxima a 98 — a requisição ao cilindro 105. Desta mesma maneira, serão atendidas as requisições 112, por ser a mais próxima a 105, e sucessivamente, 87, 140, 184, 32, 21 e 16. Utilizando este algoritmo, o deslocamento total do braço de leitura é de 304 cilindros, pouco mais da metade do deslocamento exigido pelo algoritmo anterior. Apesar de não criar escalonamentos ótimos, o SSTF diminui consideravelmente o deslocamento do braço de leitura, em comparação ao FCFS.

Como a fila de requisições pendentes é dinâmica e novas requisições podem ser inseridas a qualquer momento, o SSTF não é livre de inanição. Uma eventual requisição ao cilindro 190, criada no momento em que o cilindro 184 é lido, será atendida antes da requisição ao cilindro 32. Supondo que um fluxo contínuo de requisições próximas a 184 seja criado, o braço de leitura pode nunca atender a requisição ao cilindro 32.

2.3.3 SCAN

O algoritmo SCAN [25] previne a ocorrência de inanição nas requisições ao disco. No SCAN, o braço de leitura é posicionado em uma das extremidades do disco, deslocando-se sobre toda a superfície em direção ao lado oposto. Ao alcançar o extremo oposto, o

movimento é repetido em sentido inverso, até chegar à posição original. O movimento é interrompido a qualquer momento, em ambas as direções, para atender requisições. Devido ao comportamento do braço de leitura neste algoritmo, ele também é conhecido como *algoritmo do elevador*.

Supondo que no exemplo anterior (requisições aos cilindros 32, 16, 112, 87, 184, 105, 21 e 140, e posição inicial 98) o braço de leitura esteja deslocando-se em direção ao cilindro 0, serão servidas as requisições 87, 32, 21 e 16. Ao atender o cilindro 16, o braço de leitura finaliza seu ciclo movimentando-se em direção ao cilindro 0. No cilindro 0 o movimento é invertido, e as requisições 105, 112, 140 e 184 são servidas, nesta ordem. O deslocamento total do braço de leitura utilizando este algoritmo é de 282 cilindros. Além de diminuir o deslocamento do braço de leitura, se comparado aos dois algoritmos anteriores, o SCAN previne a ocorrência de inanição.

2.3.4 C-SCAN

Supondo que a distribuição das requisições nos cilindros de um disco seja linear, a concentração de requisições em posições nas quais o braço de leitura passou recentemente deve ser pequena. Ao chegar a um extremo do disco, por exemplo, a maior concentração de requisições deve estar próxima ao lado oposto, e não no extremo próximo ao braço de leitura, onde as requisições foram atendidas recentemente. Este é o princípio do algoritmo C-SCAN ou *Circular SCAN*, que provê tempo de acesso mais uniforme que o seu antecessor, o algoritmo SCAN. Assim como no SCAN, o C-SCAN movimenta o braço de leitura entre os extremos do disco. Entretanto, ao alcançar o lado oposto, o braço de leitura retorna a posição original, sem atender nenhuma requisição no caminho de volta.

2.3.5 LOOK e C-LOOK

Os algoritmos SCAN e C-SCAN, como descrito anteriormente, movimentam o braço de leitura sobre toda a superfície do disco. Assim, caso o movimento inicie no cilindro 0, o braço deslocar-se-á sobre todos os cilindros até alcançar o último, iniciando então o movimento inverso. Na prática, não é necessário que o braço de leitura desloque-se até as

extremidades do disco, mas somente até atender a última requisição em cada direção. No exemplo da fila de requisições do algoritmo SCAN, por exemplo, a direção do braço de leitura poderia ser invertida após atender à requisição ao cilindro 16, ao invés de deslocar-se até o cilindro 0 e voltar. Desta forma, o deslocamento total do exemplo seria reduzido a 250 cilindros. As versões do SCAN e C-SCAN que implementam essa restrição são chamadas de LOOK e C-LOOK, respectivamente.

2.4 Acesso a Disco no Kernel Linux

Atualmente, nenhum dos algoritmos clássicos de escalonamento (descritos nas Subseções anteriores) são utilizados em sua forma básica. Em geral, são utilizados como base para a construção de algoritmos mais sofisticados, devido à complexidade e as peculiaridades dos sistemas operacionais atuais. Nesta seção são discutidas questões sobre o funcionamento dos discos e dispositivos de bloco no sistema operacional Linux.

No Kernel Linux [26], todos os dispositivos de armazenamento de bloco, secundários ou terciários, são operados por um mesmo conjunto de componentes, chamado camada de bloco ou *block layer*. O *block layer* tem o objetivo de abstrair a complexidade no acesso a dispositivos de bloco através da modularização em componentes. Ainda assim, fornece uma interface padronizada para o acesso a qualquer dispositivo de bloco, bem como aumenta o desempenho no acesso, através do escalonamento e acoplamento de requisições e caches em memória principal.

A Figura 2.2 ilustra a organização dos componentes da camada de bloco. O primeiro componente desta camada é o VFS (*Virtual File System*), responsável por intermediar todas as requisições de acesso (chamadas de sistema) aos dispositivos de bloco. O VFS também mantém em *cache* informações acessadas recentemente; desta forma, ele é responsável por decidir se as informações podem ser lidas do *cache* ou se o dispositivo deve ser acessado. O VFS define ainda a interface a ser implementada por todos os sistemas de arquivos, além de encaminhar as requisições ao sistema de arquivos de destino.

O segundo componente da camada de bloco é o *Mapping Layer*, responsável pelo mapeamento entre os dados requisitados pela aplicação — através do VFS — e o endereço

físico dos dados no dispositivo. Esse mapeamento é realizado pelo sistema de arquivos que armazena os dados acessados, e depende fortemente de características como o método de alocação e gerenciamento de espaço livre, forma de implementação de diretórios e o uso de tabelas de indexação.

Após identificar o endereço físico do bloco a ser acessado, uma operação de I/O é criada. A camada responsável por gerenciar requisições de I/O é a camada genérica de blocos ou *generic block layer* (Figura 2.2). Cada operação de I/O deve especificar um ou mais blocos físicos adjacentes a serem lidos. Caso seja preciso ler blocos não adjacentes do dispositivo, várias operações de I/O são iniciadas. O *generic block layer* também abstrai particularidades dos dispositivos de bloco, oferecendo uma visão abstrata desses periféricos. Volumes lógicos como LVM (*Logical Volume Manager*) [7] e RAID (*Redundant Array of Independent Disks*) também são gerenciados por esta camada.

Cada dispositivo de bloco possui seu próprio escalonador, para onde são repassadas as requisições de I/O criadas pela camada genérica de blocos. O escalonador mantém uma fila de requisições pendentes, ordenada por setor. As requisições podem ser atendidas linearmente, voltando ao início da fila quando a última requisição for atendida (seguindo o princípio do algoritmo C-LOOK), ou de acordo com determinada política de escalonamento. Para minimizar o tempo de atendimento, o escalonador agrupa requisições a blocos adjacentes, de forma que possam ser atendidas com uma única operação. Os escalonadores possuem também uma fila de envio ou *dispatch queue*, onde são inseridas as requisições já escalonadas, na ordem em que devem ser executadas pelo *driver* do dispositivo; este responsável por realizar a comunicação com a controladora do periférico.

No Kernel Linux, o algoritmo de escalonamento pode ser especificado na inicialização do sistema operacional, pelo *driver* do dispositivo ou até mesmo em tempo de execução (via *sysfs*). O kernel possui quatro algoritmos de escalonamento de disco: *Noop*, *Deadline*, *Anticipatory* e *CFQ* (algoritmo padrão), além de prover uma interface para que outros escalonadores possam ser implementados e inseridos como módulos do sistema. Nas Subseções seguintes serão discutidos os algoritmos de escalonamento presentes no kernel.

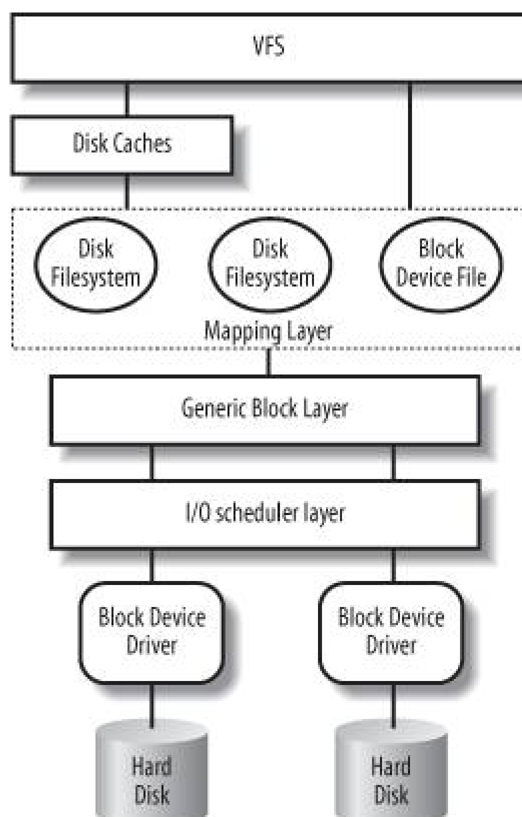

Figura 2.2: Camada de bloco do Kernel Linux [23].

2.4.1 Noop

O algoritmo *Noop* ou *No Operation* é o mais simples dos escalonadores do kernel. No *Noop*, todas as requisições são inseridas em uma mesma fila e executadas em ordem de chegada. Este escalonador é uma implementação do algoritmo FCFS. Em geral, o *Noop* é utilizado para fins didáticos, ou quando o escalonamento é realizado por algum outro componente.

2.4.2 Deadline

O algoritmo *Deadline* é basicamente a implementação de uma variação do algoritmo EDF (*Earliest Deadline First*) para o Kernel Linux. Neste algoritmo, cada requisição possui um *deadline*, que, por padrão, é igual a 500 milissegundos para operações de leitura e 5 segundos para operações de escrita. Operações de leitura têm menor *deadline* pois usualmente são síncronas, ou seja, bloqueiam os processos que as executam, enquanto operações de escrita geralmente são assíncronas. O *Deadline* mantém quatro filas de

requisições: duas ordenadas por setor, uma para operações de leitura e outra para escrita, e duas ordenadas por *deadline*, uma para leitura e outra para escrita.

Para iniciar o processo de atendimento, o escalonador primeiramente seleciona uma das filas de requisições. Se há operações de escrita e leitura pendentes, umas das filas de leitura será selecionada, ao menos que as filas de escrita já tenham sido descartadas muitas vezes (evitando inanição de operações de escrita). Se o *deadline* da primeira requisição da fila ordenada por tempo tiver expirado, esta deve ser atendida; caso contrário, a próxima requisição da fila ordenada por setor é servida. Por questões de desempenho, a cada requisição servida, um conjunto (*batch*) de requisições a setores próximos também é atendido. A existência de requisições a setores próximos é verificada através da fila ordenada por setor.

Parâmetros como o tamanho deste conjunto, assim como o *deadline* das operações de escrita e leitura, e o número máximo de vezes que operações de leitura podem ter prioridade sobre operações de escrita, podem ser alterados em tempo de execução. O algoritmo *Deadline* é recomendado para alguns tipos de *workloads*, como em sistemas de banco de dados, por exemplo [27].

2.4.3 Anticipatory

O algoritmo de escalonamento *Anticipatory* [14] (ou simplesmente *AS*) é similar ao algoritmo *Deadline*. Ele mantém as mesmas quatro filas do algoritmo anterior — duas ordenadas por setor e duas ordenadas por *deadline*. O *deadline* padrão para operações de leitura é de 125 milissegundos, enquanto que, para operações de escrita, é de 250 milissegundos. Como no escalonador *Deadline*, as requisições da fila ordenada por setor são atendidas linearmente, leitura e escrita, mas com preferência a operações de leitura. Caso o *deadline* de alguma requisição expire, ela será atendida juntamente com um conjunto de requisições a setores próximos.

O *AS* difere do algoritmo anterior em dois aspectos. Em primeiro lugar, eventualmente, o algoritmo pode atender a requisições *atrás* do braço de leitura, fazendo-o deslocar-se em sentido inverso. Isso ocorre somente quando a distância da posição atual do braço

de leitura até a posição desta requisição for menor que a metade da distância da posição atual à próxima requisição em sentido normal. Existe também um limite de distância para o movimento contrário do braço.

Em segundo lugar, após cada leitura, a próxima requisição é examinada. Se a requisição é muito próxima ou pertence ao mesmo processo da anterior, ela é executada imediatamente. Caso contrário, o escalonador examina as informações que armazena sobre o padrão de acesso de cada processo. Caso o algoritmo conclua que, de acordo com seu histórico de acesso, é provável que o processo executará outra operação de leitura, o atendimento de requisições é suspenso por um curto período (cerca de 10 milissegundos), aguardando por novas requisições deste processo. A preferência na execução de requisições subsequentes de uma mesma aplicação é devida à localidade no acesso, que beneficia o desempenho do disco.

Como os escalonadores escolhem a próxima requisição a ser executada logo após o atendimento da anterior, algumas vezes podem ser tomadas decisões equivocadas. Por exemplo, muitas aplicações executam uma grande quantidade de leituras síncronas e sequenciais, e entre cada leitura, uma pequena quantidade de processamento (alocação de memória ou incremento de contadores, por exemplo). O problema ocorre quando o escalonador assume que a aplicação não executará requisições momentaneamente, e atende requisições de outras aplicações, em outras partes do disco. O *AS* foi o primeiro algoritmo a prevenir a ocorrência deste fenômeno, chamado *deceptive idleness* [13].

2.4.4 CFQ

O CFQ (*Complete Fairness Queueing*) é um algoritmo de escalonamento de I/O baseado em algoritmos de enfileiramento justo, ou *fair queueing*, utilizados tipicamente em escalonamento de tráfego de redes [28]. Algoritmos de enfileiramento justo são utilizados quando diversos fluxos de dados precisam compartilhar um mesmo canal limitado de comunicação. No CFQ, cada processo possui uma fila, onde são inseridas as suas requisições. Diferentemente de outros escalonadores de I/O, neste algoritmo o disco é compartilhado através da alocação de *timeslices* para os processos, similar ao que ocorre

no escalonamento de CPU.

Este escalonador também introduz o conceito de prioridades de I/O, de forma que processos com maior prioridade possam preemptar (interromper) processos de prioridade menor. A prioridade de I/O de um processo deriva de sua prioridade de CPU, podendo ser alterada pelo administrador do sistema. O escalonador permite também que processos deixem o dispositivo em estado ocioso por um pequeno intervalo de tempo — somente durante o seu *timeslice* — aguardando por novas requisições deste mesmo processo, evitando a ocorrência de *deceptive idleness*. Caso não seja criada uma nova requisição durante o período ocioso, ou esta requisição resulte em um grande movimento do braço de leitura, o processo perderá temporariamente o direito de deixar o dispositivo em estado ocioso. O CFQ permite também um pequeno movimento inverso do braço de leitura, como no algoritmo *Anticipatory*.

O CFQ, que é atualmente o escalonador de I/O padrão do Kernel Linux, é baseado no algoritmo de escalonamento de I/O SFQ (*Stochastic Fair Queueing*) [29]. O SFQ também agrupa as requisições de um mesmo processo em filas, mas se limita a um número pré-definido destas estruturas, utilizando uma função *hash* para mapear as requisições de determinado processo às filas. Utilizando esta abordagem, as *hashes* de diferentes processos podem colidir, sendo mapeadas para uma mesma fila e prejudicando o desempenho e o isolamento da banda de I/O dos processos. O CFQ cria filas exclusivas para os processos de maneira dinâmica.

CAPÍTULO 3

QUALIDADE DE SERVIÇO

A Qualidade de Serviço — *Quality of Service*, ou QoS — é uma propriedade inerente à transferência de dados sobre um canal de comunicação. É a habilidade de estabelecer prioridades e garantir determinados requisitos de desempenho a um fluxo de dados em particular. Uma comunicação com QoS deve garantir desempenho previsível, e não ser afetada por condições externas, como o número de fluxos concorrentes ou a quantidade de informações por eles gerados [30, 31, 32]. Um fluxo de dados com QoS é definido por um conjunto de atributos de desempenho, dependendo das características do canal de comunicação.

A QoS surgiu da necessidade de priorizar certos fluxos de pacotes em redes de computadores, principalmente na Internet. Com o crescimento da utilização dessa rede e a demanda por aplicações multimídia, aumentou a necessidade por mecanismos que fornecessem garantias concretas de serviço, onde a simples utilização de algoritmos clássicos de controle de congestionamento não seria suficiente [31].

Alguns modelos de serviço foram propostos pela IETF (*Internet Engineering Task Force*) para atender a demanda de QoS. Historicamente, os modelos que mais se destacaram foram os serviços integrados (*IntServ*) [33] e os serviços diferenciados (*DiffServ*) [34]. O modelo de serviços integrados é caracterizado pela reserva de recursos. Antes de iniciar uma comunicação, é preciso configurar o caminho e reservar os recursos necessários para garantir a qualidade de serviço da transmissão. O protocolo RSVP (*Resource ReSerVation Protocol*) [35] especifica a troca de mensagens responsável por realizar essa alocação. Ele permite também que diversos transmissores enviem dados para vários grupos de receptores, sendo particularmente útil para implementação de mutidifusão ou *multicast*.

O modelo de serviços diferenciados é caracterizado pela definição de *classes de serviço*. Geralmente, classes de serviço são oferecidas pelo ISP (*Internet Service Provider*), e têm

diferentes requisitos e garantias, especificados em *Service Level Agreements* — SLA [4]. A classificação dos pacotes é realizada de acordo com o campo TOS (*Type Of Service*) do cabeçalho do IPv4 [32]. Por utilizar um campo de comprimento fixo, o número de classes é limitado. Uma outra estratégia para implementação de *DiffServ* é o MPLS, ou *Multi-Protocol Label Switching* [36]. O MPLS opera entre as camadas de enlace e rede do modelo OSI, e especifica um *rótulo* ou *label* para cada mensagem. O roteamento é realizado com base nos rótulos, independente do fluxo ou do protocolo que esteja encapsulado pelo pacote MPLS. A grande vantagem do *DiffServ* é a facilidade de implementação e implantação, quando comparado ao *IntServ*.

Apesar da QoS ser estudada principalmente na área de redes de computadores, os seus conceitos podem ser estendidos e aplicados a outros tipos de canais de comunicação. A proposta deste trabalho é desenvolver um algoritmo de escalonamento que forneça garantias de QoS no acesso a disco. O acesso a disco possui diversas particularidades que requerem a adaptação dos algoritmos consagrados de QoS para redes de computadores ou o desenvolvimento de novos mecanismos. Os requisitos de qualidade de serviço relevantes a este trabalho são: largura de banda (*bandwidth*), latência (*delay*) e rajadas (*burst*).

A largura de banda é o atributo que limita a quantidade de informação que pode ser transmitida por um fluxo de dados em determinado intervalo de tempo. No caso do acesso a disco, quanto maior a largura de banda reservada, mais requisições serão atendidas por unidade de tempo. Um dos problemas ao garantir largura de banda em acesso a disco, é que a mesma depende fortemente do padrão de acesso das aplicações. Por exemplo, para garantir a mesma largura de banda a duas aplicações, *Al* e *Seq*, onde esta realiza acesso sequencial, e aquela, acesso aleatório, é necessário reservar o disco para *Al* por um intervalo de tempo maior do que a *Seq*.

Outro atributo importante em QoS é a latência ou *delay*. A latência é a quantidade de tempo decorrido desde o envio da informação, até o recebimento pelo destinatário. Não existe qualquer relação entre largura de banda e latência em um canal de comunicação. No escopo deste trabalho, assumiremos que a latência é o tempo decorrido desde a criação de uma requisição até a entrega dos dados à aplicação.

Um último atributo também muito utilizado em qualidade de serviço é a rajada (*burst*). Opcionalmente, as rajadas podem ser subdividas em dois atributos: tamanho máximo de rajada e duração máxima de rajada. As rajadas acontecem após um fluxo permanecer sem utilizar o canal de comunicação por determinada quantidade de tempo. Quando este fluxo finalmente utilizar o recurso, é garantida uma largura de banda superior à que havia sido reservada — não superior ao tamanho máximo de rajada. Esta rajada pode se estender por um período definido como a duração máxima de rajadas, voltando à largura de banda anterior após este intervalo.

Embora não abordada por este trabalho, outro importante atributo de QoS é a flutuação, também chamada de *jitter*. Conceitualmente, flutuação é a variação da latência entre transmissões. Não existe, também, uma forte relação entre latência e flutuação, de forma que determinados tipos de aplicação podem ser sensíveis a este, mas não àquele. Aplicações de *streaming* de áudio ou vídeo, por exemplo, são sensíveis à flutuação, mas não à latência. Mesmo que as transmissões de um fluxo de *streaming* sejam realizadas com alta, porém constante latência (baixa flutuação), não haverá prejuízo na execução do fluxo [32]. No caso do acesso a disco, pode haver situação semelhante quando aplicações multimídia utilizam constantemente o disco. Para atenuar o efeito da flutuação, geralmente são utilizados *buffers*. Quanto maior o tamanho do *buffer*, menor é o efeito da flutuação sobre o fluxo. Contudo, o controle dos *buffers* foge ao escopo deste trabalho por ser realizado em nível de aplicação. Nas Seções seguintes serão discutidos alguns algoritmos clássicos de QoS.

3.1 Algoritmo do Balde Furado

O algoritmo do balde furado, ou *leaky bucket* [37], é um sistema de enfileiramento com taxa de saída constante. Este algoritmo é baseado na abstração de um balde com um pequeno furo no fundo, de forma que o fluxo de escoamento da água pelo furo ocorra a uma taxa constante, ρ, independente da quantidade e da taxa de entrada de água no balde, ou à taxa zero caso o balde esteja vazio. Além disso, caso seja inserida mais água do que a capacidade do balde, a água transbordará e não fará parte do fluxo de saída,

isto é, será descartada.

A mesma ideia pode ser aplicada a mensagens em um fluxo de comunicação. O balde é representado por uma fila de tamanho previamente definido. Ao chegar uma nova mensagem, se houver espaço, ela será inserida na fila, caso contrário será descartada. Em intervalos regulares de tempo, uma mensagem é retirada do início da fila e transmitida.

Esse algoritmo é muito utilizado em redes de computadores, para transformar fluxos irregulares de pacotes em fluxos regulares de saída. Um exemplo de utilização é a transmissão de dados por uma interface de rede. Em sistemas convencionais podem existir diversas aplicações com taxas irregulares de criação de pacotes. Estas taxas irregulares podem ser suavizadas pelo algoritmo, reduzindo o efeito de rajadas e diminuindo a possibilidade de congestionamento na rede [32].

3.2 Algoritmo do Balde de Símbolos

O algoritmo do balde de símbolos, mais conhecido como *token bucket*, impõe um padrão de saída rígido à taxa média, independente das irregularidades do tráfego de entrada [32]. Para tal, é utilizada a mesma abstração do algoritmo anterior — um balde — mas neste caso símbolos (ou *tokens*) são inseridos a uma razão constante, até o limite de capacidade do balde. Para que uma mensagem seja transmitida através do canal de comunicação, um símbolo deve ser *consumido*, ou, caso o balde esteja vazio, esperar até que mais símbolos sejam inseridos. Este algoritmo é utilizado quando se deseja permitir o envio de rajadas. De certa forma, o balde de símbolos permite que entidades *poupem* capacidade de transmissão, até um determinado limite, para o posterior envio de rajadas. Outra diferença fundamental entre os dois algoritmos é que o balde de símbolos pode descartar *tokens* (capacidade de transmissão) caso o balde esteja cheio, mas não pacotes, como o algoritmo do balde furado. Este algoritmo pode ser implementado através de um simples contador de *tokens*, incrementado a uma razão constante, e decrementado a cada envio de mensagem.

A duração máxima das rajadas permitidas pelo algoritmo depende da capacidade de armazenamento de *tokens* do balde. Supondo que uma entidade possua o balde cheio

de símbolos, ela poderá enviar mensagens até que se esgotem seus *tokens*. Utilizando apenas este algoritmo, não é possível estabelecer limites para a taxa de envio de rajadas, possibilitando que um fluxo utilize toda a capacidade do canal e prejudique as garantias de QoS de outros fluxos que compartilhem o mesmo recurso. Para contornar esta situação, uma das estratégias utilizada é acrescentar um balde furado à saída do balde de símbolos, com taxa de escoamento igual a taxa máxima permitida pelas rajadas.

A combinação desses algoritmos, balde furado e de símbolos, permite a modelagem de diversos tipos de problemas. Existem muitas estratégias derivadas destes dois algoritmos, como por exemplo o *Hierarchical Token Bucket*, ou HTB [38], uma abordagem hierárquica de baldes de símbolos, utilizado para a distribuição de largura de banda de rede entre diferentes classes de aplicações.

3.3 Algoritmos de Enfileiramento Justo

Em qualquer canal de comunicação compartilhado onde não haja mecanismos adequados de controle, um fluxo poderá utilizar mais do que a sua parcela da capacidade. Se um fluxo requisitar massivamente a utilização do canal, por exemplo, e a política de atendimento for a ordem de chegada, é provável que as garantias de QoS dos demais sejam defasadas. Visando atender a este tipo de problema, o primeiro algoritmo de enfileiramento justo, ou *fair queueing*, foi proposto [28]. Mais tarde, todos os algoritmos dele derivados também ficaram conhecidos como *algoritmos de enfileiramento justo*. Enquanto os algoritmos dos baldes, discutidos nas Seções anteriores, regulam a taxa média de transmissão e de rajadas, os algoritmos de enfileiramento justo garantem o compartilhamento do recurso entre diversos fluxos. Apesar de semelhantes, têm objetivos distintos.

No algoritmo de enfileiramento justo inicial, cada fluxo de um canal compartilhado possui uma fila, onde são armazenados os pacotes por ele enviados. Quando ocioso, o canal de comunicação percorrerá estas filas, enviando um pacote de cada fluxo, de forma que todos enviem o mesmo número de pacotes. Um dos problemas deste algoritmo é que, caso um fluxo agrupe diversos pacotes pequenos em pacotes maiores, ele poderá utilizar uma fração maior da capacidade do canal que um fluxo que envie os mesmos pacotes

separadamente, pois este algoritmo contabiliza apenas o número de pacotes, não os seus tamanhos.

Hoje, existem diversos algoritmos de enfileiramento que se aproximam do resultado obtido pelo GPS (*General Processor Sharing*) [39] na divisão da capacidade de um canal de comunicação. O GPS é uma disciplina de enfileiramento que realiza o compartilhamento proporcional de um link, de forma que todo fluxo receba exatamente a sua parte, proporcional ao seu *peso*, em determinado intervalo de tempo. Entretanto, como o GPS assume a comunicação como um fluxo contínuo de bits (pacotes muito pequenos) e que vários fluxos podem utilizar o canal simultaneamente, não é possível utilizá-lo na prática, sendo usado apenas teoricamente para medir a corretude de algoritmos de enfileiramento. Alguns algoritmos de enfileiramento justo foram propostos com o objetivo de aproximar o resultado obtido pelo GPS, dentre os quais destacam-se: WFQ [40], $\mathrm{WF^2Q}$ [41] e $\mathrm{WF^2Q+}$ [42], descritos a seguir.

A primeira aproximação do GPS baseada em pacotes foi chamada de PGPS ou *Packet General Processor Sharing*, ficando também conhecida como *Weighted Fair Queueing* ou WFQ. Neste algoritmo são atribuídas duas *tags* (ou *timestamps*) a todos os pacotes, que representam os instantes de início e término de envio do pacote na disciplina GPS. Tais *tags* são denominadas *start tag* e *finish tag*, respectivamente. Em determinado instante, o pacote com menor *finish tag* dentre todos os pacotes pendentes do sistema, ou seja, o próximo pacote que teria o envio finalizado na disciplina ideal GPS, é selecionado para envio. Além disso, o WFQ também previne que um fluxo seja beneficiado por agrupar diversos pacotes pequenos em pacotes maiores. Isto se deve ao fato de que, no WFQ, a *finish tag* é calculada proporcionalmente ao tamanho em *bytes* do pacote, não apenas pela quantidade de pacotes enviados [41].

Para exemplificar o funcionamento do WFQ, consideremos o sistema com 11 fluxos concorrentes apresentado na Figura 3.1, onde o eixo vertical ilustra os diferentes fluxos e o horizontal, o tempo. Neste exemplo, o primeiro fluxo tem peso igual a 0,5; os demais, 0,05. Por simplicidade, assumimos que todos os pacotes têm tamanho 1, e que 1 é a velocidade do canal de comunicação. O i-ésimo pacote do fluxo n é representado por P_n^i.

No referido exemplo, o primeiro fluxo envia 11 pacotes subsequentes, enquanto o restante dos fluxos envia apenas 1 pacote no instante 0, conforme mostra a Figura 3.1 (a). Considerando a disciplina ideal GPS, são necessárias 2 unidades de tempo para transmitir pacotes do primeiro fluxo e 20 para transmitir pacotes dos demais[1]. A Figura 3.1 (b) ilustra a ordem de atendimento dos pacotes deste exemplo pela disciplina GPS.

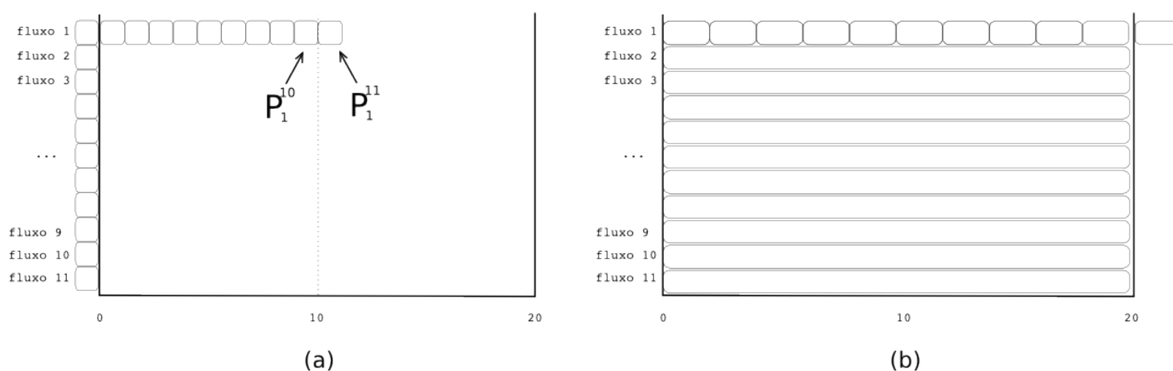

Figura 3.1: (a) Ordem de envio de pacotes e (b) ordem de atendimento pela disciplina GPS.

A Figura 3.2 (a) mostra a ordem de atendimento dos pacotes do exemplo anterior pelo algoritmo WFQ. Como descrito, as *tags* atribuídas aos pacotes representam os instantes de início e término do atendimento pelo GPS — o que pode ser observado na Figura 3.1 (b). Deste modo, como o atendimento dos pacotes é baseado nas *finish tags*, os 10 primeiros pacotes do fluxo 1 serão atendidos antes de quaisquer pacotes de outros fluxos. Além disso, há um grande intervalo de tempo entre o envio do décimo e do décimo primeiro pacote deste fluxo. Neste exemplo, além de causar grande latência nos pacotes dos fluxos com peso 0,05, a flutuação dos pacotes do fluxo 1 também é alta.

Conforme ilustrado pelo exemplo, é constatado que o algoritmo WFQ pode causar oscilações indesejáveis no tempo de envio de pacotes. Com o objetivo de diminuir estas oscilações, um novo algoritmo, conhecido como WF^2Q, ou *Worst-case Fair Weighted Fair Queueing*, foi proposto. O WF^2Q é semelhante ao seu predecessor, WFQ, diferindo em apenas um aspecto: ao invés de selecionar o pacote com menor *finish tag* entre todos os pacotes do sistema, são considerados apenas os pacotes cujo atendimento na disciplina

[1] O tempo de transmissão de um pacote é obtido através do quociente entre o tamanho do pacote e o produto do peso do fluxo pela capacidade do canal de comunicação.

GPS já fora iniciado (*start tag* menor que o tempo atual). Esses pacotes são denominados *elegíveis*. A Figura 3.2 (b) ilustra a ordem de atendimento dos pacotes do exemplo anterior utilizando este algoritmo. Após atender o primeiro pacote do fluxo 1, é necessário que o sistema atenda um dos pacotes dos demais fluxos, pois o próximo pacote do fluxo 1, apesar de ter a menor *finish tag*, ainda não é elegível.

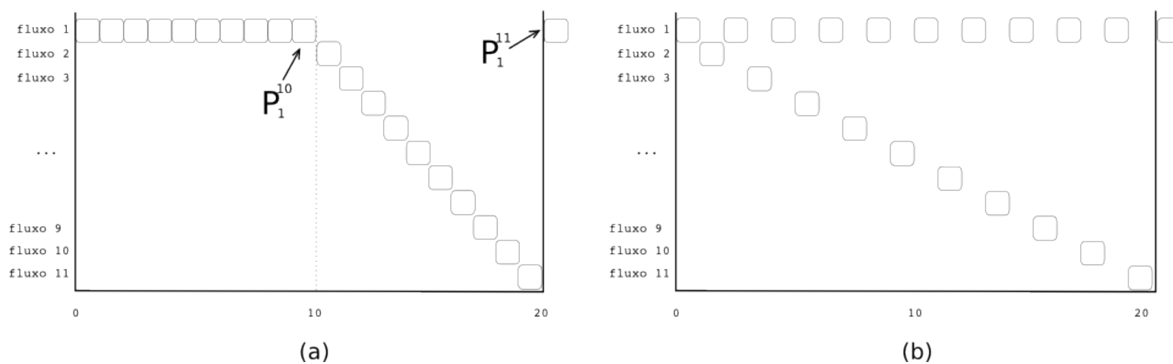

Figura 3.2: Ordem de atendimento dos algoritmos (a) WFQ e (b) WF^2Q.

Em [42], é proposta uma otimização no algoritmo WF^2Q. O algoritmo resultante, chamado WF^2Q+, apesar de utilizar as mesmas políticas de atribuição de *tags* e envio de pacotes de seu antecessor, reduz substancialmente a complexidade na computação do próximo fluxo a ser atendido.

CAPÍTULO 4

TRABALHOS RELACIONADOS

A abordagem atualmente mais utilizada para fornecer garantias de qualidade de serviço no acesso a disco é o desenvolvimento de novos algoritmos de escalonamento. Os trabalhos realizados nesta área podem ser classificados em dois principais grupos: 1) escalonadores de disco de tempo real e 2) escalonadores de disco baseados em algoritmos de enfileiramento justo (*fair queueuing*).

Os escalonadores de tempo real atribuem um *deadline* para cada requisição, reorganizando as requisições pendentes em função deste atributo. Em geral as requisições são atendidas de acordo com o seu *deadline*, priorizando as que possuem o deadline mais próximo. A utilização estrita desta política de escalonamento no acesso a disco, também conhecida como EDF ou *Earliest Deadline First* [43], resulta em perda de desempenho, devido ao *seek-time* e a latência de rotação causados. Desta forma, foram propostas modificações a este esquema básico de escalonamento.

O algoritmo PSCAN, ou *Priority*-SCAN [44], classifica as requisições em níveis de prioridade de acordo com seus *deadlines*. Após atender uma requisição, é verificada a existência de requisições pendentes em níveis mais altos de prioridade. Caso seja encontrada, o escalonador passa a atender as requisições deste nível. Requisições de um mesmo nível são atendidas de acordo com a política SCAN.

Em [45], outro algoritmo de escalonamento de tempo real é proposto: FD-EDF ou *Feasible Deadline* EDF. Neste algoritmo, as requisições cujo *deadline* seja viável são atendidas em ordem EDF. Um *deadline* é considerado viável se é estimado que ele possa ser atendido a tempo. Uma requisição cujo *deadline* seja considerado não viável pode ser atendida somente quando não existirem mais requisições com *deadline* viável. O FD-EDF é baseado no algoritmo FD-SCAN [46], que segue o mesmo princípio; contudo, atende requisições com *deadline* viável em ordem SCAN.

Por último, o algoritmo SCAN-EDF [47] atende requisições em ordem EDF, utilizando a política SCAN quando há mais de uma requisição com o mesmo *deadline*. O objetivo é garantir as restrições de tempo e ao mesmo tempo utilizar o disco de forma eficiente. Na prática, a eficiência deste algoritmo depende da quantidade de requisições com o mesmo *deadline*. Alguns mecanismos foram sugeridos para aumentar este número [48].

Em todos os algoritmos de escalonamento de disco de tempo real descritos, é possível configurar a latência das requisições através do seu *deadline*. Entretanto, nenhum deles possui controle sobre outros atributos de QoS, como largura de banda e rajadas. Além disso, nenhum dos algoritmos anteriores implementa mecanismos de prevenção de *deceptive idleness*.

Algoritmos de enfileiramento justo foram utilizados inicialmente no contexto de redes de computadores, com o objetivo de alocar de forma justa a capacidade de um canal de comunicação compartilhado entre diversos fluxos de dados [32]. Dentre os algoritmos clássicos de enfileiramento justo, destacam-se: WFQ, WF^2Q e WF^2Q+ (descritos na Seção 3.3). Nos escalonadores de disco baseados nesta classe de algoritmos, são atribuídas duas *tags* ou *timestamps* a todas as requisições, baseadas em tempo real ou virtual, denominadas tag de início (*start tag*) e tag de término (*finish tag*). Finalmente, as requisições são escalonadas de acordo com essas *tags*, proporcionando boa distribuição da banda disponível [40, 41, 42]. YFQ [11], EYFQ [49], BFQ [12] e pClock [5] são alguns exemplos de algoritmos de disco baseados em enfileiramento justo.

O algoritmo YFQ (*Yet Another Fair Queueing*) atende requisições em lotes. Para que um lote de requisições seja servido, é necessário que todas as requisições do lote anterior tenham finalizado o atendimento. Como na seleção do próximo conjunto de requisições são consideradas todas as aplicações que possuem requisições pendentes, é pouco provável que um mesmo lote possua várias requisições de uma mesma aplicação, particularmente se essa aplicação cria requisições síncronas. Este comportamento reduz o desempenho do disco, pois não executa requisições com localidade. Além disso, por ser baseado no algoritmo WFQ, é possível configurar apenas o atributo peso (*weight*) das aplicações, não sendo possível configurar atributos como largura de banda, latência e rajadas de forma

direta.

Em [49] é proposto um novo algoritmo de escalonamento baseado no YFQ, denominado EYFQ ou *Extended Yet Another Fair Queueing*. O EYFQ estende o YFQ em três aspectos: 1) inclusão de um algoritmo de controle de admissão, 2) criação de mecanismos de monitoramento de desempenho e ajuste de peso baseado em retroalimentação (*feedback*) e 3) suporte a alocação da banda de disco ociosa. Apesar de introduzir funcionalidades interessantes ao algoritmo YFQ, as garantias de QoS fornecidas são as mesmas do seu predecessor.

Os algoritmos BFQ e pClock, que proveem as garantias de QoS mais próximas às do HTBS, proposto por este trabalho, são descritos em detalhes nas Seções seguintes.

4.1 Budget Fair Queueing

O BFQ, ou *Budget Fair Queueing*, é um escalonador baseado em algoritmos de enfileiramento justo, que possibilita a alocação de uma fração da capacidade do disco a cada aplicação, proporcional ao seu peso. Quando enfileiradas, as aplicações recebem determinada quantidade de *budgets*, que representa a quantidade de setores de disco que elas poderão transferir. Essa disciplina de alocação de recursos também é conhecida como *Token Bucket*, ou balde de símbolos [32].

Uma vez selecionada, uma aplicação terá acesso exclusivo ao disco, até que se esgotem seus *budgets* ou suas requisições pendentes. Em seguida, uma nova aplicação será escolhida de acordo com o algoritmo de enfileiramento justo chamado *Budget*-WF^2Q+, ou BWF^2Q+, uma extensão do algoritmo WF^2Q+. Para evitar a ocorrência de inanição, existe um limite máximo e configurável de *budgets* que uma aplicação pode armazenar, representado por B_{max}.

Diferentemente de outros escalonadores, o BFQ pode atender requisições de uma mesma aplicação em lotes, cujo tamanho é igual ao número de *budgets* da aplicação em determinado instante. Este esquema aumenta o desempenho do disco, devido à localidade do acesso e possibilita o agrupamento de requisições. Contudo, esse comportamento só é possível em aplicações que utilizem requisições assíncronas.

No BFQ, o disco é mantido ocioso por um pequeno instante após o atendimento de requisições síncronas, evitando a ocorrência de *deceptive idleness*. Por este motivo, ele é classificado como um algoritmo não-conservativo. Essa otimização, presente também nos algoritmos Anticipatory [14] e CFQ [15], garante boa utilização da capacidade do disco, visto que a maioria dos acessos a disco é feita de forma sequencial e síncrona [12].

Entretanto, assim como em outros escalonadores baseados em algoritmos de enfileiramento justo, no BFQ não é possível configurar a largura de banda e latência de forma independente, nem o suporte a rajadas.

4.2 pClock

O pClock é um algoritmo de escalonamento baseado em curvas de serviço, ou *service curves* [50]. Tais curvas são definidas através de três atributos: largura de banda, latência e rajadas, que controlam os parâmetros do *Token Bucket* de cada aplicação e as *tags* atribuídas às requisições.

Para cada requisição de acesso são atribuídas duas tags: a tag de início, que representa o momento em que a requisição foi criada, e a tag de término, que representa o seu *deadline*. A alocação de banda de disco é controlada através da quantidade de *tokens* inserida em um balde, de forma que o atributo rajada limite a sua quantidade máxima de *tokens*. As requisições são então atendidas de acordo com suas tags de término.

O pClock, por ser um escalonador conservativo, possui mecanismos de alocação de banda de disco ociosa entre aplicações, não as penalizando pela utilização adicional do disco. A banda ociosa também pode ser alocada a aplicações em *background*, conhecidas como aplicações *best-effort*. É provado que aplicações que cumpram suas curvas de serviço, ou seja, não excedam os limites preestabelecidos, nunca perderão seus *deadlines*, independentemente dos padrões de acesso de outras aplicações [5].

Para que seja possível atender os requisitos de desempenho de todas as aplicações, o pClock define matematicamente um limite mínimo para a largura de banda que o sistema de armazenamento deve possuir. Entretanto, é muito difícil estimar a capacidade real de um disco, pois esta depende de inúmeros parâmetros, como o padrão de acesso, localidade

e até posição dos dados na superfície do disco [11].

Como no algoritmo pClock o atendimento das requisições é baseado exclusivamente no *deadline*, o algoritmo não se beneficia da localidade das requisições, o que pode resultar na subutilização da largura de banda do disco. Ainda, como o algoritmo é conservativo, pode haver *deceptive idleness* na presença de requisições sequenciais e síncronas.

4.3 Comparação entre os Algoritmos

A Tabela 4.1 compara o algoritmo HTBS, proposto neste trabalho, com escalonadores que proveem garantias de QoS semelhantes. Como os algoritmos de tempo real (*real-time*) descritos neste Capítulo possuem basicamente as mesmas garantias de QoS, eles são agrupados em uma única coluna. O mecanismo de alocação de banda de disco adotado pelo algoritmo CFQ é baseado em prioridades, ou seja, aplicações com maiores prioridades percebem maior largura de banda. Tanto o algoritmo YFQ quanto o BFQ alocam a banda de disco através de frações de disco, também chamadas de peso, o que vincula a largura de banda percebida pela aplicação à capacidade total do disco. Dos algoritmos listados, apenas o pClock e o HTBS possibilitam a configuração direta da largura de banda, em IOPS (I/O por segundo) ou KB/s (*kilobytes* por segundo).

Algoritmos	**Real-time**	**CFQ**	**YFQ**	**BFQ**	**pClock**	**HTBS**
Alocação de banda	Não	Prioridade	Peso	Peso	Sim	Sim
Controle de latência por aplicação	Sim	Não	Não	Não	Sim	Sim
Controle de rajadas	Não	Não	Não	Não	Sim	Sim
Prevenção de *deceptive idleness*	Não	Sim	Não	Sim	Não	Sim

Tabela 4.1: Comparação com outros algoritmos.

CAPÍTULO 5

HIGH-THROUGHPUT TOKEN BUCKET SCHEDULER

Este trabalho propõe o HTBS (*High-throughput Token Bucket Scheduler*), um novo algoritmo de escalonamento que utiliza conceitos retirados de algoritmos anteriores — BFQ e pClock. O objetivo é utilizar partes destes algoritmos para fornecer garantias de QoS, causando o menor impacto possível na performance do disco. Tomamos inicialmente o algoritmo pClock, descrito na Seção 4.2, por prover bom isolamento de performance (como mostrado nos experimentos presentes em [5]), além da possibilidade de ajustar os parâmetros largura de banda, rajadas e latência de forma independente. Apesar disso, o algoritmo pClock apresenta baixo desempenho na presença de requisições síncronas, como ilustrado pelos experimentos descritos no Capítulo 6.

Diferentemente do pClock, o HTBS é não-conservativo, pois implementa as políticas de prevenção de *deceptive idleness* do algoritmo BFQ, descrito na seção 4.1. Por este motivo, algumas das garantias matemáticas expostas em [5] referentes ao algoritmo pClock não se aplicam diretamente ao novo algoritmo. Contudo, mostramos através de experimentos que é possível manter as garantias relativas ao isolamento de performance e ao mesmo tempo aumentar o desempenho do disco.

Com relação ao algoritmo BFQ, apesar de fornecer alto desempenho na presença de requisições síncronas, que são a maioria nos sistemas atuais [12], não implementa mecanismos que possibilitem a configuração de garantias de QoS de forma individual a aplicações. Em suma, o HTBS incorpora a ideia de prevenção de *deceptive idleness* adota pelo BFQ, mesclada à política de atribuição de *tags* e atendimento de requisições do algoritmo pClock.

A Seção 5.1 apresenta definições e uma visão geral do problema tratado pelo algoritmo. O HTBS é apresentado e explicado em detalhes na Seção 5.2, enquanto considerações a respeito de seu funcionamento e parametrização são discutidas em 5.3.

5.1 Visão Geral

O objetivo deste trabalho é definir um novo algoritmo de escalonamento que garanta tanto isolamento de performance como alto desempenho. Assumimos neste trabalho que um sistema de armazenamento compartilhado é formado por três componentes, ilustrados na Figura 5.1: 1) um disco, capaz de atender requisições de leitura e escrita a um ou mais blocos de armazenamento contíguos, chamados *setores*; 2) um escalonador de requisições de disco; e 3) n filas de requisições, referentes às n aplicações competindo pelo sistema. Aplicações são quaisquer entidades que possam utilizar o disco, como processos, grupos de processos, usuários, grupos de usuários, *threads* ou máquinas virtuais. Do ponto de vista do escalonador, o disco é composto por uma grande e única sequência de setores, numerados de forma crescente.

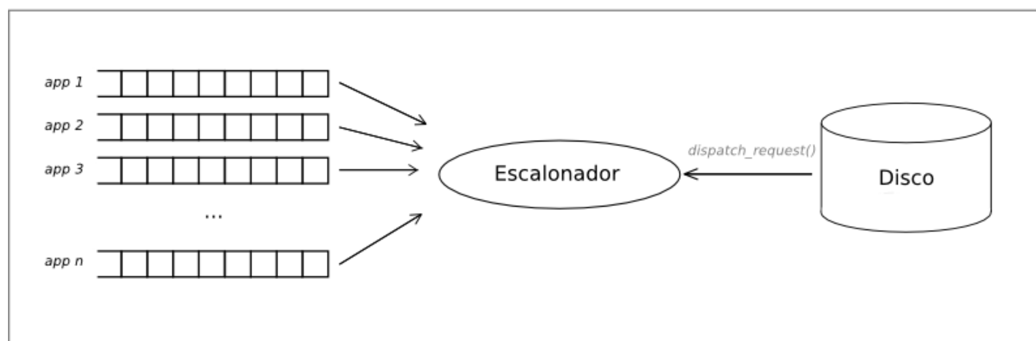

Figura 5.1: Modelagem do sistema.

Como definido em [5], uma *reserva* de disco é composta por uma das n filas e seus três atributos de desempenho, σ_i, ρ_i e δ_i, sendo σ_i o tamanho máximo de rajadas, ρ_i a largura de banda e δ_i o limite máximo de latência da i-ésima reserva. O *backlog* de uma reserva i, representado por B_i, é o número de requisições pendentes, ou seja, requisições aguardando atendimento. Uma reserva i está *backlogged* se $\mathrm{B}_i > 0$. É denominada *ativa* a reserva que atualmente está sendo servida pelo disco. No restante deste trabalho, assumimos que os termos *reserva*, *aplicação* e *fluxo* referem-se à definição de reserva discutida neste parágrafo.

Duas requisições são ditas sequenciais se a posição do final de uma requisição é adjacente ao início da próxima; caso contrário elas são denominadas não-sequenciais ou

aleatórias. Caso a emissão de uma requisição dependa do término da anterior, elas são denominadas síncronas. De forma análoga, requisições assíncronas não possuem relações de dependência. De acordo com [12], definimos T_{wait} como o tempo máximo que o escalonador aguardará ociosamente por nova requisição síncrona e sequencial de uma mesma reserva. Por último, B_{max} limita o número de requisições síncronas consecutivas que podem ser servidas a uma reserva.

Um escalonador que nunca desperdice capacidade, ou seja, nunca mantenha o disco ocioso enquanto houver requisições pendentes, é dito conservativo, ou *work-conserving*. Por outro lado, caso o escalonador mantenha o disco ocioso mesmo existindo requisições pendentes, ele é denominado não-conservativo, ou *non-work-conserving*. Embora paradoxal, manter o disco ocioso por curtos intervalos de tempo pode aumentar significativamente o desempenho em alguns casos. Isto ocorre porque a maioria das aplicações atuais utilizam requisições síncronas intercaladas com pequenas quantidades de processamento, como, por exemplo, alocação ou liberação de memória, processamento dos dados obtidos ou atualização de variáveis de controle [12].

Caso não haja uma nova requisição imediatamente após o término da anterior, um escalonador conservativo passará a atender uma nova aplicação. Essa constante troca de atendimento entre aplicações é prejudicial ao desempenho do disco, porque ocasiona *seek time* e latência de rotação em excesso, devido à falta de localidade no acesso. Esta perda de desempenho é denominada *deceptive idleness* (ociosidade enganosa) [13], e uma aplicação que se encontre neste tempo entre requisições é dita *deceptively idle* (enganosamente ociosa).

Escalonadores não-conservativos podem manter o disco ocioso por um curto intervalo de tempo após o término de uma requisição, com o objetivo de aguardar o processamento da aplicação e a chegada de uma nova requisição. Caso uma nova requisição seja criada neste intervalo de tempo, ela será atendida imediatamente; caso contrário, a aplicação poderá perder o direito de manter o disco em estado ocioso. Esse é o mecanismo de prevenção de *deceptive idleness* utilizado neste trabalho.

5.2 O Algoritmo

O algoritmo proposto neste trabalho é baseado em *tags* ou *timestamps*. Toda requisição encaminhada ao HTBS é repassada a uma de suas reservas, dependendo da aplicação que a criou. Cada requisição criada recebe duas *tags*: a tag de início ou *start tag* (S_i^j), e a tag de término ou *finish tag* (F_i^j), onde i representa a reserva e j o identificador da requisição. A tag de início representa o momento de chegada da requisição no escalonador (como mostrado adiante, caso uma aplicação exceda suas garantias de QoS, pode ser adicionado um *delay* a suas tags de início), enquanto a tag de término representa o seu *deadline*, ou seja, a tag de início somada ao atributo de latência de sua reserva ($S_i^j + \delta_i$). Além disso, cada reserva i possui duas outras tags: $MinS_i$, que representa a menor tag de início entre as requisições pendentes de i e $MaxS_i$ que representa a soma da maior tag de início em i e $1/\rho_i$.

O algoritmo principal é mostrado em pseudo-código na Figura 5.2. A próxima requisição a ser atendida é selecionada pela função *dispatch_request*, enquanto *add_request* adiciona novas requisições. A função *dispatch_request* (linha 7) realiza dois procedimentos: definir a aplicação ativa (*active_app*) e a partir desta selecionar uma requisição pendente para atendimento. A aplicação ativa é alterada somente em três casos: 1) a aplicação ativa anterior atingiu o limite de requisições consecutivas B_{max}, 2) não foram criadas novas requisições no intervalo de tempo T_{wait} ou 3) o padrão de acesso da aplicação não é sequencial. Quando houver necessidade de alteração, será selecionada a aplicação que possuir a requisição pendente com menor tag de término (linhas 10 e 11).

Em seguida, caso a aplicação ativa esteja *backlogged* ($B_i > 0$), será atendida sua requisição pendente com menor tag de término (linha 14). Caso contrário, o disco será mantido ocioso por no máximo T_{wait} milissegundos (linha 16), aguardando novas requisições desta aplicação, evitando a ocorrência de *deceptive idleness*. Assim como no algoritmo BFQ, a função *set_timer* é responsável por iniciar o período de ociosidade do disco; *unset_timer*, por interrompê-lo. Se após a chamada da função *set_timer* (linha 16) T_{wait} milissegundos passarem sem que a aplicação ativa crie novas requisições, a função *timer_expired* (linha 20) é chamada, forçando a escolha de uma nova aplicação ativa.

```
1  add_request (i, r)
2    if active_app == i and i is waiting for the next request then
3       unset_timer ()
4    update_num_tokens ()
5    check_and_adjust_tags ()
6    compute_tags ()

7  dispatch_request ()
8    if active_app == nil or
9    active_app dispatched more than B_max then
10      w = request with minimum finish tag F_j^w
11      active_app = application j who issued w
12   else
13      if active_app is backlogged then
14         w = request with minimum finish tag F_j^w from active_app
15      else
16         set_timer (T_wait)
17         return nil
18   MinS_k = S_k^w
19   return w

20 timer_expired ()
21   active_app = nil
22   dispatch_request ()
```

Figura 5.2: Algoritmo principal do HTBS.

Quando uma nova requisição é criada, através da função *add_request* (linha 1), existem dois cenários possíveis. Se o disco está sendo mantido ocioso e a requisição pertence à aplicação ativa, o algoritmo executa a função *unset_timer* (linha 3), prevenindo a chamada da função *timer_expired*. Desta forma, a requisição recém-criada será selecionada imediatamente para atendimento na próxima execução da função *dispatch_request*. Por outro lado, se o disco não estava sendo mantido ocioso, a nova requisição é enfileirada junto a outras requisições da mesma aplicação. Em ambos os casos, três funções baseadas no algoritmo pClock são executadas: *update_num_tokens*, *check_and_ajust_tags* e *compute_tags*.

A Figura 5.3 apresenta essas três funções, chamadas sempre que a função *add_request* é executada. A função *update_num_tokens* atualiza o número de tokens de uma reserva. Os novos tokens disponibilizados são proporcionais ao tempo decorrido desde a última atualização, bem como a quantidade de banda ρ alocada para a reserva (linha 3). Tokens regulam a quantidade de requisições que podem ser criadas por uma aplicação, sem que os atributos de desempenho sejam violados. Esta função também controla as rajadas, representadas pelo atributo por σ, por limitar a quantidade máxima de tokens armazenados em uma reserva (linhas 4 e 5).

```
1  update_num_tokens ()
2    Let Δ be the time interval since last request
3    numtokens_i += Δ x ρ_i
4    if numtokens_i > σ_i then
5        numtokens_i = σ_i

6  check_and_adjust_tags ()
7    Let C be the set of all backlogged reservations
8    if ∀ j ∈ C, MinS_j > t_r then
9        mindrift = min_{j ∈ C} {MinS_j − t_r}
10       ∀ j ∈ C, subtract mindrift from MinS_j, MaxS_j and all start and finish tags

11 compute_tags ()
12   if numtokens_i < 1 then
13       S_i^r = max {MaxS_i , t}
14       MaxS_i = S_i^r + 1 / ρ_i
15   else
16       S_i^r = t
17   F_i^r = S_i^r + δ_i
18   numtokens_i −= 1
```

Figura 5.3: Funções que controlam a atribuição de *tags* e *tokens*.

Tags são atribuídas a novas requisições através da função *compute_tags*. A menos que a aplicação exceda os seus atributos de desempenho, a tag de início de uma requisição

será igual ao tempo atual (linha 16). Caso a aplicação ultrapasse os seus atributos de desempenho, isto é, o seu número de tokens esteja negativo, à tag de início será atribuído um valor maior que o tempo atual (linha 13). Na prática, atribuir um tempo futuro à tag de início, tem como objetivo aproximar o valor que esta mesma tag teria se a aplicação criasse o mesmo número de requisições sem exceder os limites estabelecidos. A tag de término sempre corresponde à soma do valor da tag de início e do atributo de latência da reserva (linha 17) .

Por último, a função *check_and_adjust_tags* evita que fluxos sejam penalizados pela utilização da banda ociosa do disco. Como explicado, requisições de aplicações que excedam seus atributos recebem tags de início no futuro. Desta forma, quanto mais banda ociosa uma aplicação utilizar, mais distante do tempo atual serão suas tags de início. Seja um sistema com três reservas *a*, *b* e *c*, por exemplo, onde *a* e *b* estejam ociosas e *c* esteja utilizando mais do que seus atributos de desempenho. Como *c* está excedendo seus limites, suas tags de início estarão cada vez mais distantes no futuro. Quando *a* e *b* voltarem a criar requisições, suas requisições terão tags de início no tempo atual, fazendo com que as requisições da aplicação *c*, com tags de início no futuro, sofram inanição. Para impedir que isso ocorra, caso todas as tags *MinS* de todas as aplicações *backlogged* estejam no futuro (linha 8), a menor diferença entre *MinS* e o tempo atual (linha 9) é subtraído de todas as tags das aplicações *backlogged* (linha 10).

5.3 Parâmetros

O HTBS possui dois parâmetros ajustáveis: T_{wait}, que limita a quantidade de tempo que o disco pode ser mantido ocioso enquanto estiver aguardando novas requisições da aplicação ativa, e B_{max}, que representa o número máximo de requisições consecutivas que uma aplicação pode executar.

O valor adequado para T_{wait} depende fortemente do sistema e das características de cada aplicação. É necessário aumentar este intervalo caso as aplicações realizem uma quantidade maior de processamento entre requisições. Contudo, atribuir um valor muito alto para T_{wait} pode diminuir o desempenho do disco, pois também será alto o tempo

máximo que o disco poderá ser mantido ocioso desnecessariamente, no caso de uma aplicação ter finalizado o seu acesso ao disco, por exemplo. Além disso, caso o tempo que uma aplicação leve para criar sua próxima requisição seja grande o suficiente, pode ser mais eficiente atender uma requisição pendente de outra aplicação, mesmo perdendo localidade. O valor ideal para T_{wait} é o menor tempo necessário para as aplicações síncronas criarem suas próximas requisições, sem causar grande impacto no desempenho do sistema. Nos experimentos realizados, o valor utilizado para T_{wait} é de 10 milissegundos.

O número máximo de requisições síncronas e consecutivas que o disco atenderá de uma mesma aplicação é regulado por B_{max}. Quanto maior o valor de B_{max}, maior será o número de requisições executadas com localidade, sendo que a localidade é um dos fatores com maior influência no desempenho do disco. Entretanto, como o disco atenderá exclusivamente requisições de uma aplicação se o valor de B_{max} for alto, pode ocorrer inanição de requisições de outras aplicações. Da mesma forma, quanto maior o valor de B_{max}, mais defasadas serão as garantias de latência e maior será a flutuação (*jitter*) percebida. Na prática, quanto maior o valor de B_{max}, maior a largura de banda do disco, mas maior também é a latência média entre as requisições. O atributo deve ser configurado de forma adequada, ponderando entre alto desempenho e baixa latência.

CAPÍTULO 6

RESULTADOS EXPERIMENTAIS

Para a realização dos experimentos, os algoritmos HTBS e pClock foram implementados como módulos para o Kernel Linux 2.6.35 [16]. Para o algoritmo BFQ, foi utilizada a implementação oficial que pode ser obtida em [51]. Todos os testes foram executados em um PC equipado com um processador AMD Athlon X2 240, 2800 MHz, *dual-core*, e 4 GB de memória RAM DDR3. O disco utilizado é um Samsung HD080HJ SATA, 80 GB, 7200 rpm e 8 MB de cache *onboard*, sem suporte a NCQ (*Native Command Queueing*) [52].

Todos os *workloads* utilizados nos experimentos são sintéticos, gerados pela ferramenta de benchmark fio [17]. O fio permite a realização de testes de benchmark de workloads específicos de I/O, sendo possível especificar parâmetros como a API utilizada para as requisições (síncrona ou assíncrona), padrão de acesso (sequencial ou aleatório), tipo da requisição (leitura ou escrita), tamanho das requisições, taxa de criação de requisições, dentre outros parâmetros. Esta ferramenta permite também a execução de *jobs* concorrentes com parametrização individual.

A seguir são apresentados os resultados de quatro experimentos: 1) comparação do algoritmo proposto com o algoritmo pClock na presença de duas aplicações com diferentes atributos de desempenho, 2) impacto do parâmetro B_{max} na latência das requisições, 3) utilização de rajadas e 4) comparação da largura de banda acumulada utilizando diversas aplicações.

6.1 Comparação com pClock

A principal diferença entre o HTBS e o pClock é que, no algoritmo proposto, foram criados mecanismos para que as garantias de QoS não fossem prejudicadas por aplicações síncronas, pois, segundo [12], grande parte das requisições em sistemas reais são síncronas. Neste primeiro experimento são criadas duas aplicações (*jobs*): *app1* com largura de banda

igual a 200 KB/s e latência 50 milissegundos, e *app2*, com largura de banda de 800 KB/s, e latência 100 milissegundos. *App1* tem o padrão de acesso aleatório, enquanto *app2* é sequencial; ambas síncronas. O experimento foi executado por 5 minutos, com requisições de 4k e B_{max} igual a 20.

As Figuras 6.1 e 6.2 mostram a largura de banda alcançada pelas aplicações durante o teste, onde o eixo vertical representa a largura de banda (medida em KB/s) e o eixo horizontal mostra o tempo em milissegundos. De acordo com o gráfico da Figura 6.1, o algoritmo HTBS alocou, em média, a quantidade de largura de banda esperada para *app1* e *app2*, 200 KB/s e 800 KB/s, respectivamente. Já na Figura 6.2 é possível observar que os requisitos de QoS de ambas as aplicações não foram atendidos, pois a mesma quantidade de banda foi alocada para *app1* e *app2*; cerca de 280 KB/s.

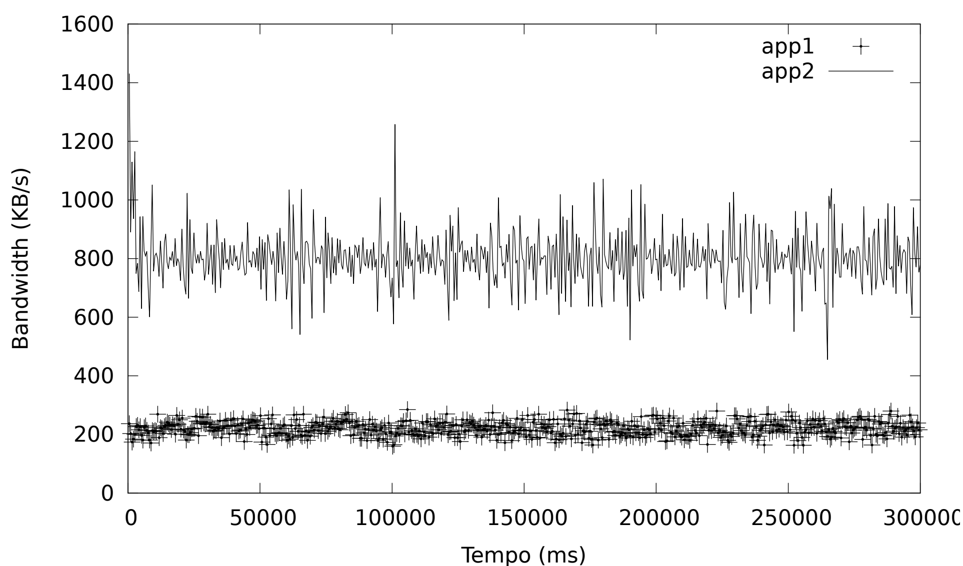

Figura 6.1: Largura de banda para o algoritmo HTBS.

Neste experimento, o algoritmo pClock além de não cumprir o requisito de largura de banda de *app2*, alocou uma quantidade maior de banda para *app1* do que o esperado. Com efeito, mesmo não alocando a quantidade de banda adequada à *app2*, *app1* foi capaz de utilizar banda de disco ociosa. Isto ocorre pois, por serem síncronas, logo após o término do atendimento a uma requisição, o *backlog* de sua respectiva aplicação estará vazio. Há um intervalo de tempo entre o término de uma requisição e a criação da próxima, durante o qual o escalonador, por ser conservativo, atenderá uma requisição pendente de outra

aplicação. Logo, o escalonador pClock atenderá as aplicações em política semelhante à *round-robin* neste cenário, prejudicando o desempenho total do disco e provendo serviço semelhante independente dos atributos de QoS atribuídos.

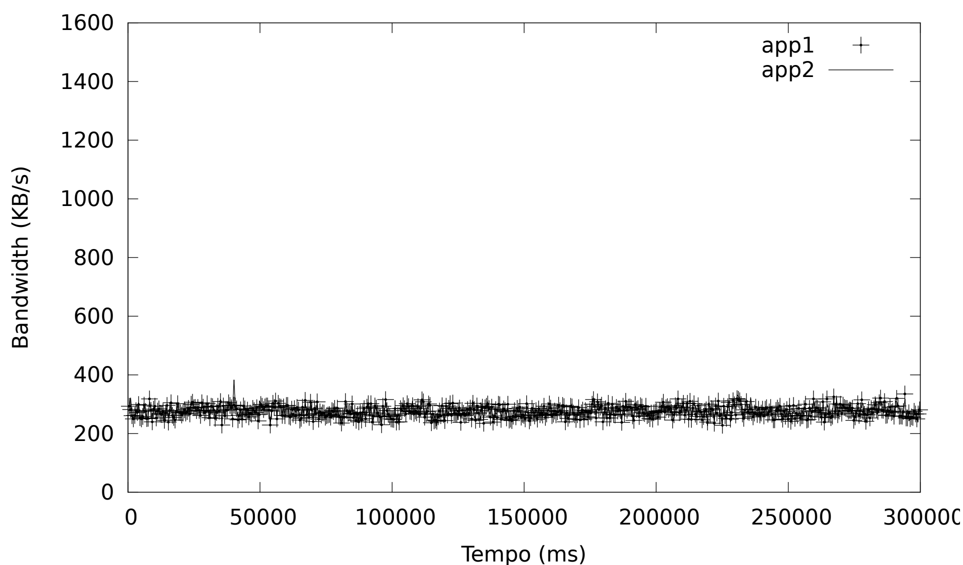

Figura 6.2: Largura de banda para o algoritmo pClock.

Como o algoritmo HTBS aguarda a chegada de novas requisições de uma mesma aplicação após o atendimento de uma requisição síncrona — o chamado mecanismo de prevenção de *deceptive idleness* — ele é capaz de tomar melhores decisões quanto à troca ou não da aplicação ativa. É importante ressaltar que, como o algoritmo BFQ não possui controle explícito de largura de banda e latência por aplicação, não é possível realizar o mesmo teste comparando HTBS e BFQ.

6.2 Impacto do Atributo B_{max} na Latência

O atributo B_{max} limita a quantidade máxima de requisições consecutivas atendidas de uma mesma aplicação. Se por um lado quanto maior o valor do atributo maior o benefício pela localidade no acesso, por outro, um valor muito grande pode prejudicar as garantias de latência e até causar inanição. Neste experimento é analisado o impacto do atributo B_{max} sobre as garantias de latência do algoritmo.

Como no experimento anterior, são criadas duas aplicações com os mesmos padrões de acesso e garantias de QoS — *app1*, aleatória, largura de banda igual a 200 KB/s e latência

50 milissegundos, e *app2*, sequencial, 800 KB/s de largura de banda e 100 milissegundos de latência. Os gráficos apresentados nas Figuras 6.3 e 6.4 mostram a latência sofrida pelas requisições de ambas as aplicações quando executadas com B_{max} igual a 1 (Figura 6.3) e B_{max} igual a 20 (Figura 6.4). Na prática, utilizar o algoritmo HTBS com B_{max} igual a 1 é equivalente a utilizar o algoritmo pClock; logo, este experimento compara a latência sofrida por ambos os algoritmos. O tempo total de execução do teste é de 10 segundos.

A Figura 6.3 mostra a latência sofrida pelas requisições para B_{max} igual a 1. Neste caso, a latência média de ambas as aplicações é de 14 milissegundos, enquanto o desvio padrão é igual a 5,2 milissegundos para *app1* e 5,7 milissegundos para *app2*. Para B_{max} igual a 20, ilustrado na Figura 6.4, a latência média observada é de 21,6 milissegundos para app1 e 4,5 para app2. Da mesma forma, o desvio padrão é igual a 10 milissegundos para *app1* e 7,4 milissegundos para *app2*.

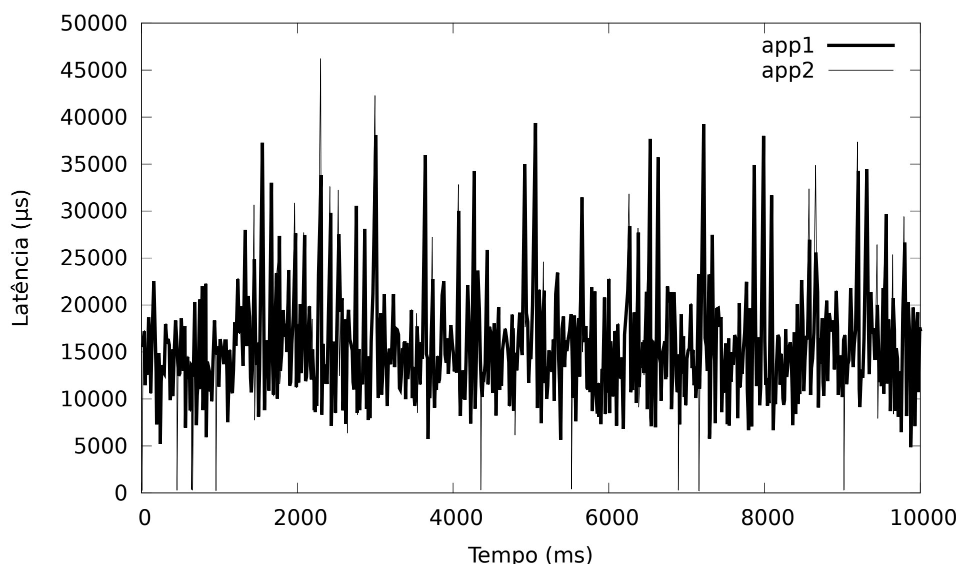

Figura 6.3: Latência utilizando o algoritmo HTBS com $B_{max} = 1$.

Neste experimento é possível observar que *app1* tem um pequeno aumento em sua latência média com o aumento de B_{max}, pois até 20 requisições de *app2* podem ser atendidas antes que a sua próxima requisição pendente seja servida. Este fato também influenciou a diminuição na latência média das requisições de *app2* embora o desvio padrão tenha aumentado, pois existem valores baixos para a latência, quando são atendidas requisições consecutivas, e valores altos, quando a aplicação ativa é alterada, causando *seek time*.

Mesmo assim, apesar do aumento no desvio padrão da latência das requisições, nenhuma requisição de *app1* ou *app2* estourou o seu deadline neste experimento.

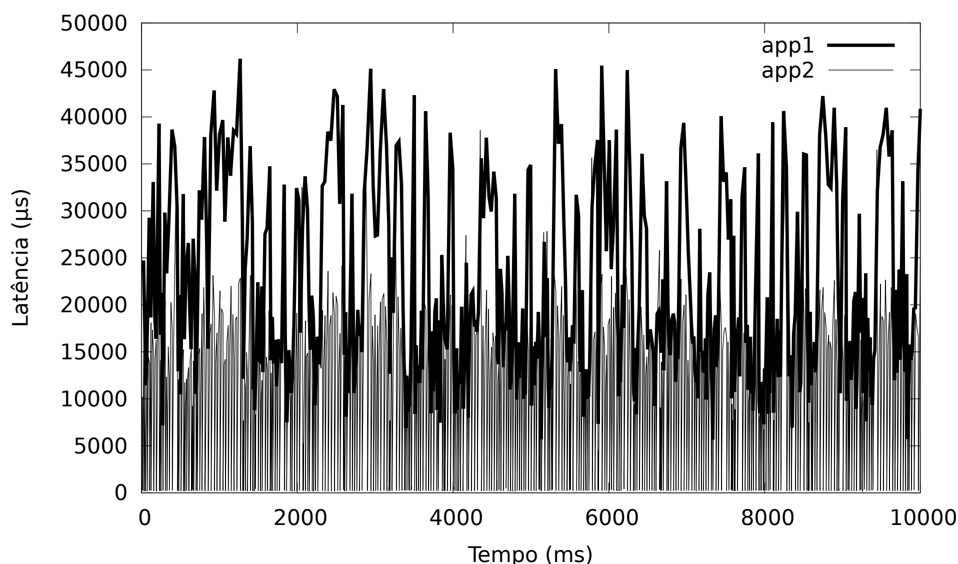

Figura 6.4: Latência utilizando o algoritmo HTBS com $B_{max} = 20$.

6.3 Utilização de Rajadas

Este experimento tem o objetivo de comparar o comportamento dos algoritmos HTBS, pClock e BFQ na presença de rajadas. O suporte a rajadas deve ocorrer nos casos em que, após permanecer ociosa ou utilizando menos que a sua largura de banda reservada por um intervalo de tempo, uma aplicação requisite uma quantidade de banda maior que o estipulado. Nestes casos, é desejável que o algoritmo de escalonamento permita que a aplicação exceda seus atributos por um curto período, sem prejudicar as garantias de QoS das demais aplicações.

Para testar o comportamento dos algoritmos foi criado um caso de teste contendo duas aplicações executando leituras sequenciais, *app1* e *app2*. Ambas as aplicações possuem os mesmos atributos de QoS: largura de banda de 400 KB/s e latência 100 milissegundos. Após 10 segundos de execução, outra aplicação é iniciada, *app3*, com largura de banda igual a 4000 KB/s, rajadas de 4000 KB e latência de 100 milissegundos. O tempo total de execução do teste é de 30 segundos.

O resultado do experimento com o algoritmo pClock é ilustrado pela Figura 6.5. Novamente, como o algoritmo não possui mecanismos para o tratamento de requisições síncronas, e em determinado instante cada aplicação possui apenas uma requisição pendente, o atendimento é semelhante ao da política *round-robin*. Assim, mesmo diante do início de *app3*, após 10 segundos de teste, *app1* e *app2* continuam utilizando mais do que sua largura de banda (400 KB/s), não possibilitando o uso de rajadas por *app3*.

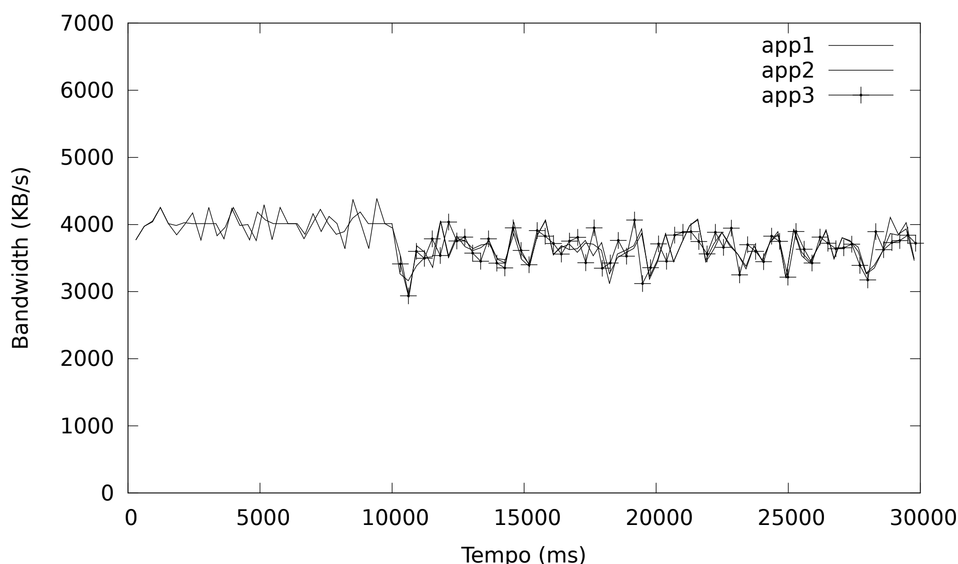

Figura 6.5: Rajadas utilizando o algoritmo pClock.

A Figura 6.6 apresenta o resultado do caso de teste utilizando o algoritmo BFQ. Como pode ser observado, após 10 segundos de teste rajadas são alocadas à *app3*, que percebe largura de banda de aproximadamente 4900 KB/s durante 5 segundos. Logo após, decorridos 15 segundos de teste, a largura de banda de *app3* é ajustada de acordo com sua largura de banda reservada, 4000 KB/s. Devido a maneira como as *tags* são atribuídas pelo algoritmo BWF^{2}Q+, rajadas são alocadas mesmo que o algoritmo não possibilite explicitamente a sua configuração. Este comportamento diminui a flexibilidade na alocação dos recursos do sistema, pois não possibilita a configuração do tamanho máximo das rajadas, ou mesmo desabilitá-las.

Por último, a Figura 6.7 mostra o resultado obtido pelo algoritmo HTBS. Após os 10 primeiros segundos, durante os quais as duas primeiras aplicações estavam utilizando

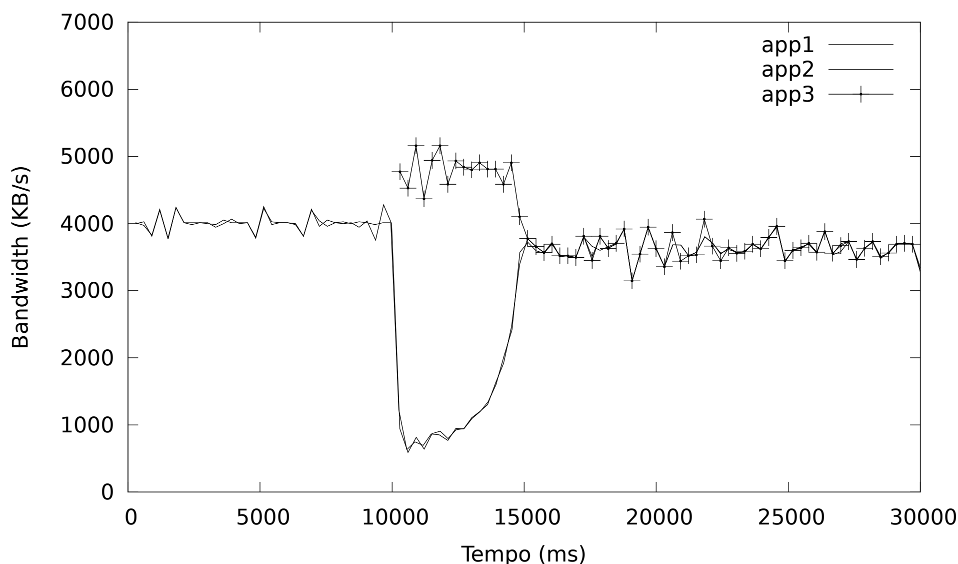

Figura 6.6: Rajadas utilizando o algoritmo BFQ.

banda de disco ociosa, quando a terceira aplicação é iniciada, o seu balde de *tokens* está com capacidade máxima, enquanto os baldes das duas primeiras aplicações estão vazios (na prática, negativos). Por este motivo, até que os *tokens* sobressalentes sejam consumidos, *app3* pode utilizar mais do que sua largura de banda. Pode também ser observado que, mesmo durante a rajada de *app3*, as garantias de QoS das demais aplicações são cumpridas.

Através deste experimento é possível verificar que o algoritmo HTBS atende o requisito de QoS rajada mesmo diante de requisições síncronas, diferentemente do que ocorre com o algoritmo pClock. Além disso, no HTBS as rajadas podem ser configuradas de forma independente, através de um dos atributos de QoS configuráveis por aplicação. No BFQ as rajadas são alocadas implicitamente devido a maneira como as *tags* são atribuídas, sem flexibilidade para adequar-se a diferentes cenários.

6.4 Largura de Banda Acumulada

Neste experimento apresentamos a largura de banda total acumulada por cada escalonador. O objetivo deste teste é analisar o desempenho dos algoritmos de escalonamento em dois cenários: utilizando-se apenas aplicações sequenciais e utilizando-se aplicações

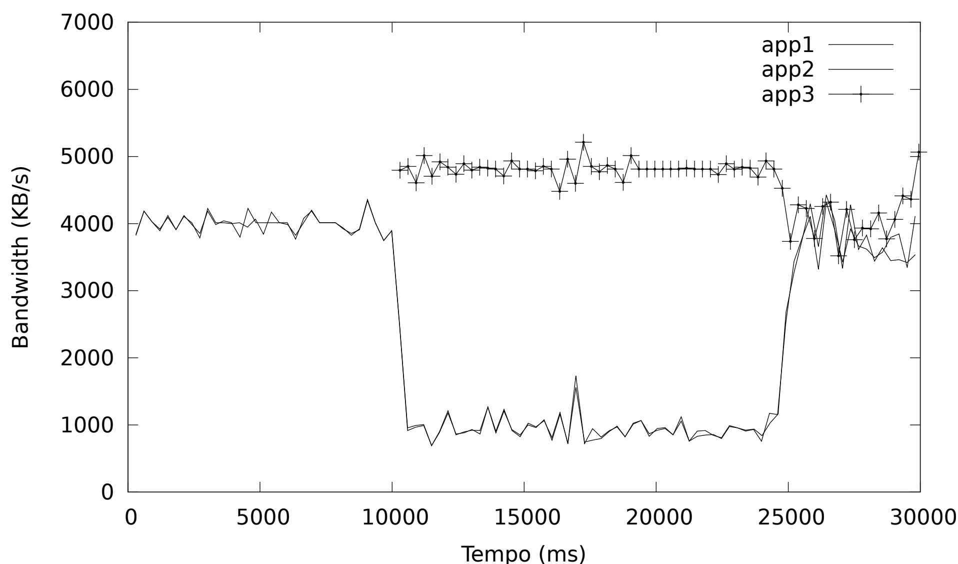

Figura 6.7: Rajadas utilizando o algoritmo HTBS.

sequenciais mescladas a aplicações aleatórias. A leitura sequencial é o padrão de acesso predominante em servidores Web, enquanto o acesso aleatório é comumente utilizado em SGBDs.

Para isso, foram criados três casos de teste, sendo cada teste executado com os três algoritmos de escalonamento — pClock, BFQ e HTBS. Todos os casos de teste são compostos por sete aplicações executando leituras síncronas e sequenciais em diferentes posições do disco. A diferença entre os casos de teste consiste em que o primeiro executa somente as sete aplicações sequenciais; o segundo executa duas aplicações aleatórias concorrentemente às sete aplicações sequenciais; no terceiro, sete aplicações sequenciais e quatro aleatórias. Mesclar gradualmente aplicações aleatórias a aplicações sequenciais tem o intuito de comparar o impacto causado por aplicações aleatórias em cada escalonador.

Cada teste foi executado por 30 segundos e o resultado mostrado corresponde à média aritmética de três execuções. Todas as aplicações aleatórias foram limitadas no *benchmark* a enviar requisições à taxa de 40 KB/s, enquanto as sequenciais foram mantidas ilimitadas. Como aplicações aleatórias degradam a performance do disco, estas foram mantidas limitadas para que o mesmo número de requisições aleatórias fossem executadas em todos os testes por todos os escalonadores. Assim, apenas o número de atendimentos de

requisições sequenciais varia entre os testes. Como o algoritmo BFQ não possui controle explícito para atributos de desempenho, a todas as aplicações foram atribuídos os mesmos parâmetros: largura de banda ilimitada (salvo requisições aleatórias) e 100 milissegundos de latência. Para os algoritmos BFQ e HTBS, o valor de B_{max} utilizado é 20.

A Figura 6.8 apresenta os resultados encontrados. No primeiro caso de teste, representado pelo *0* no eixo horizontal, como não há requisições aleatórias, o sistema tem o melhor ganho de desempenho com os algoritmos que implementam controle de *deceptive idleness*: cerca de 25%. Este ganho é devido ao fato de que os mecanismos de prevenção de *deceptive idleness* aumentam o desempenho quando existe localidade no acesso da aplicação, o que ocorre somente em aplicações sequenciais.

No segundo teste, onde existem duas aplicações aleatórias mescladas às sequenciais, pode ser observado que mesmo na presença de requisições aleatórias, como ainda é grande o número de requisições sequenciais, HTBS e BFQ apresentam desempenho superior. Entretanto, como os mecanismos de prevenção de *deceptive idleness* falham na presença de requisições aleatórias, a diferença no desempenho dos algoritmos pClock e HTBS cai para aproximadamente 18%.

No último teste, com quatro aplicações aleatórias, representadas pelo 4 no eixo horizontal, é perceptível a degradação do desempenho do sistema. Neste caso, o HTBS mostra desempenho semelhante ao pClock, pois o ganho decorrido das execuções seguidas de requisições sequenciais é pequeno se comparado ao *overhead* causado pelas requisições aleatórias. Vemos também neste experimento que a performance do HTBS é, em média, semelhante ao BFQ, mesmo que o BFQ não possua os mecanismos de controle por aplicação presentes nos outros dois algoritmos.

Através dos experimentos descritos, foi mostrado que o HTBS possui garantias semelhantes às do algoritmo pClock com relação a largura de banda, latência e rajadas. Além disso, também é mostrado que no algoritmo HTBS, diferentemente do pClock, as garantias não são afetadas por requisições síncronas, amplamente utilizadas em situações reais de uso. O algoritmo BFQ, que possui os mesmos mecanismos para o tratamento de requisições síncronas mesclados a mecanismos de isolamento de performance, não possui

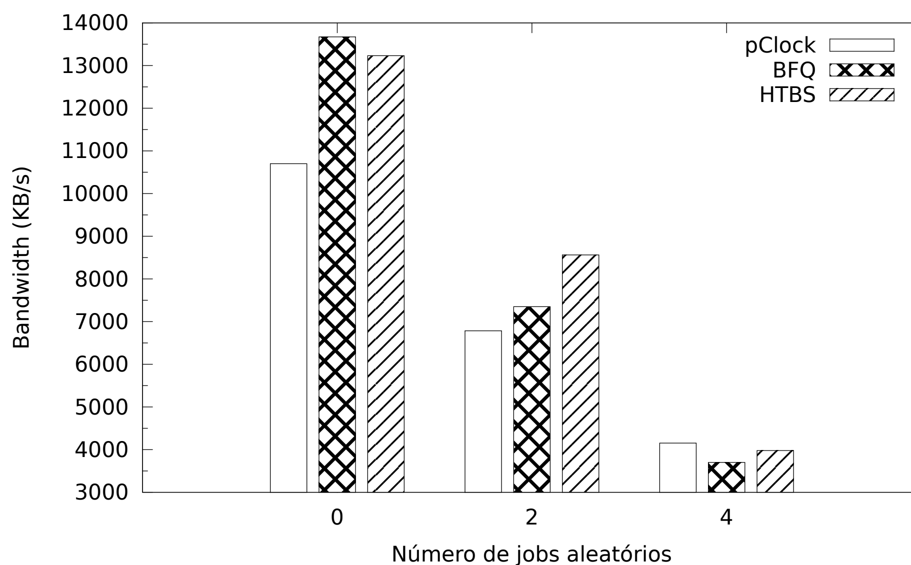

Figura 6.8: Largura de banda acumulada para os algoritmos pClock, BFQ e HTBS.

controle para os atributos largura de banda, latência e rajadas por aplicação. Mesmo assim, mostramos neste experimento que a performance do HTBS é semelhante à apresentada pelo BFQ.

CAPÍTULO 7

CONCLUSÃO

A centralização da capacidade de armazenamento em servidores de disco torna necessária a utilização de mecanismos para o gerenciamento e alocação desses recursos. Particularmente no acesso a disco, é desejável que tais mecanismos forneçam *isolamento de performance*, de modo que cada aplicação receba determinada quantidade do recurso, mensurada através de atributos de QoS, independentemente da carga do sistema.

Este trabalho apresentou o HTBS, um novo algoritmo de escalonamento de disco, que, mesclando características de dois algoritmos propostos anteriormente (pClock e BFQ), tem o objetivo de fornecer isolamento de performance a aplicações concorrentes com diferentes atributos de QoS sem que o desempenho do disco seja degradado. Através de experimentos, é mostrado que o algoritmo pClock não cumpre as garantias de QoS especificadas na presença de requisições síncronas e sequenciais, utilizadas por grande parte das aplicações atuais. Por isso, o HTBS incorpora o mecanismo para o tratamento deste tipo de requisição presente no algoritmo BFQ.

O HTBS possibilita a configuração dos seguintes atributos de QoS por aplicação: largura de banda, latência e rajadas, denominados ρ_i, δ_i e σ_i, respectivamente, onde i representa a i-ésima aplicação. Além disso, outros dois parâmetros configuráveis são a) B_{max}, que limita a quantidade máxima de requisições consecutivas atendidas de uma mesma aplicação, e b) T_{wait}, que regula o tempo máximo que o escalonador aguardará ociosamente pela próxima requisição, desde que síncrona e sequencial, de uma mesma aplicação.

Os experimentos realizados com o protótipo implementado para o Kernel Linux mostram que o novo algoritmo aumenta o desempenho geral do disco, se comparado ao algoritmo pClock, na presença de requisições síncronas e sequenciais, sem que as garantias de latência sejam violadas. Também é mostrado que o HTBS possui desempenho próximo

ao BFQ, que apesar de prevenir a ocorrência de *deceptive idleness* não possui mecanismos para a definição dos atributos largura de banda e latência por aplicação, ou o suporte a rajadas.

Dando continuidade ao trabalho, serão criados mais casos de teste com o objetivo de simular a execução de aplicações reais, como servidores Web, SGBD's e servidores de e-mail. O comportamento do novo algoritmo quando utilizado em conjunto com sistemas de arquivos, bem como a sua integração com ferramentas como LVM e RAID, será analisado para que possa ser adaptado e utilizado em servidores de armazenamento de grande porte.

BIBLIOGRAFIA

[1] C. Lumb, A. Merchant, e G. Alvarez. Façade: Virtual storage devices with performance guarantees. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, páginas 131–144. USENIX Association, 2003.

[2] R. Buyya, C. Yeo, e S. Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities. *Proceedings of the IEEE International Conference on High Performance Computing and Communications*, volume 10, páginas 5–13, 2008.

[3] B. Hayes. Cloud computing. *Communications on ACM*, volume 51, páginas 9–11. ACM, 2008.

[4] L. Vaquero, L. Rodero-Merino, J. Caceres, e M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Computer Communication Review*, volume 39, páginas 50–55. ACM, 2009.

[5] A. Gulati, A. Merchant, e P. Varman. pClock: An arrival curve based approach for QoS in shared storage systems. *In Proceedings of ACM SIGMETRICS*, páginas 13–24. ACM, 2007.

[6] S. Seelam e P. Teller. Fairness and performance isolation: an analysis of disk scheduling algorithms. *IEEE International Conference on Cluster Computing*, páginas 1–10. IEEE, 2006.

[7] LVM. LVM2 resource page. `http://sourceware.org/lvm2/`. Acessado em 14 de Abril, 2011.

[8] K. Jian, Z. Dong, N. Wen-wu, Z. Jun-wei, H. Xiao-ming, Z. Jian-gang, e X. Lu. A performance isolation algorithm for shared virtualization storage system. *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage*, páginas 35–42. IEEE Computer Society, 2009.

[9] W. Joel e A. Scott. Storage access support for soft real-time applications. *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, páginas 164–172. IEEE Computer Society, 2004.

[10] M. Andrews, M. Bender, e L. Zhang. New algorithms for the disk scheduling problem. *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, páginas 580–589, 1996.

[11] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, e A. Silberschatz. Disk scheduling with quality of service guarantees. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, páginas 400–405. IEEE Computer Society, 1999.

[12] P. Valente e F. Checconi. High throughput disk scheduling with fair bandwidth distribution. *IEEE Transactions on Computing*, volume 59, páginas 1172–1186. IEEE Computer Society, 2010.

[13] S. Iyer. The effect of deceptive idleness on disk schedulers. Dissertação de Mestrado, Rice University, 2003.

[14] S. Iyer e P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. *18th ACM Symposium on Operating Systems Principles*, páginas 117–130, 2001.

[15] J. Axboe. Linux block I/O - present and future. *Proceedings of the Ottawa Linux Symposium*, páginas 51–61, 2004.

[16] Linux Kernel Organization. The Linux Kernel Archives. `http://www.kernel.org/`. Acessado em 14 de Abril, 2011.

[17] J. Axboe. Fio - Flexible I/O tester. `http://freshmeat.net/projects/fio`. Acessado em 14 de Abril, 2011.

[18] A. Silberschatz, P. Galvin, e G. Gagne. *Operating System Concepts*. Wiley, 7 edition, 2005.

[19] R. Bez, E. Camerlenghi, A. Modelli, e A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, volume 91, páginas 489–502, 2003.

[20] C. Morimoto. *Hardware, o Guia Definitivo*. GDH Press e Sul Editores, 2nd edition, 2007.

[21] James Bottomley. Kernel Korner - Using DMA. `http://www.linuxjournal.com/article/7104`. Acessado em 6 de Junho, 2011.

[22] Steve Apiki. I/O Virtualization and AMD's IOMMU. `http://developer.amd.com/documentation/articles/pages/892006101.aspx`. Acessado em 6 de Junho, 2011.

[23] D. Bovet e M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2005.

[24] M. Hofri. Disk scheduling: FCFS vs. SSTF revisited. *Communications on ACM*, volume 23, páginas 645–653. ACM, 1980.

[25] R. Geist e S. Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Compututer Systems*, volume 5, páginas 77–92. ACM, 1987.

[26] Linux Kernel Organization. Linux kernel documentation. `http://www.kernel.org/doc/Documentation/`. Acessado em 14 de Abril, 2011.

[27] A. Jacobsen. Implementing and testing the APEX I/O scheduler in linux. Dissertação de Mestrado, University of Oslo, 2007.

[28] J. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, páginas 435–438. IEEE, 1987.

[29] P. Mckenney. Stochastic fairness queuing. 1991.

[30] X. Xiao e M. Ni. Internet QoS: A big picture. *IEEE Network*, páginas 401–404. IEEE, 1999.

[31] P. Ferguson e G. Huston. *Quality of Service*. Wiley Computer Publishing, 1998.

[32] A. Tanenbaum. *Computer Networks*. Prentice-Hall, 4th edition, 2002.

[33] R. Braden, D. Clark, e S. Shenker. Integrated services in the internet architecture: an overview. Relatório técnico, IETF, Network Working Group, 1994. RFC 1633.

[34] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, e W. Weiss. An architecture for differentiated service. Relatório técnico, IETF, Network Working Group, 1998. RFC 2475.

[35] L. Zhang, S. Berson, S. Herzog, e S. Jamin. Resource ReSerVation Protocol (RSVP) – version 1 functional specification. Relatório técnico, IETF, Network Working Group, 1997. RFC 2205.

[36] E. Rosen, A. Viswanathan, e R. Callon. Multiprotocol label switching architecture. Relatório técnico, IETF, Network Working Group, 2001. RFC 3031.

[37] J. Turner. New directions in communications (or which way to the information age). *IEEE Communications Magazine*, páginas 8–15. IEEE, 1986.

[38] J. Valenzuela, A. Monleon, I. Esteban, e O. Sallent. A hierarchical token bucket algorithm to enhance QoS in IEEE 802.11b: Proposal, implementation and evaluation. *IEEE Vehicular Technology Conference*, volume 4, páginas 2659–2662, 2004.

[39] A. Parekh e R Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, volume 1, páginas 344–357, 1993.

[40] A. Demers, S. Keshav, e S. Shenker. Analysis and simulation of a fair queueing algorithm. *Symposium Proceedings on Communications Architectures & Protocols*, páginas 1–12. ACM, 1989.

[41] J. Bennet e H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. *Proceedings of IEEE INFOCOM*, volume 1, páginas 120–128. IEEE, 1996.

[42] J. Bennet e H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networkig*, volume 5, páginas 675–689. IEEE Press, 1997.

[43] C. Liu e J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, volume 20, páginas 46–61. ACM, 1973.

[44] M. Carey, R. Jauhari, e M. Livny. Priority in DBMS resource scheduling. *Proceedings of the Fifteenth International Conference on Very Large Databases*, páginas 397–410. Morgan Kaufmann Publishers Inc., 1989.

[45] S. Chen, J. Stankovic, J. Kurose, e D. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Journal of Real-Time Systems*, volume 3, páginas 307–336, 1991.

[46] R. Abbott e H. Garcia-Molina. Scheduling I/O requests with deadlines: a performance evaluation. *Proceedings of the Real-time Systems Symposium*, volume 12, páginas 113–124, 1990.

[47] A. Reddy, J. Wyllie, e K. Wijayaratne. Disk scheduling in a multimedia I/O system. *Proceedings of ACM Multimedia*, páginas 225–233. ACM Press, 1993.

[48] J. Yee e P. Rangan. Disk scheduling policies for real-time multimedia applications. Relatório técnico, University of California, Berkeley, 1992.

[49] J. Ke, X. Zhu, W. Na, e L. Xu. AVSS: An adaptable virtual storage system. *IEEE/ACM International Symposium on Cluster Computing and the Grid*, páginas 292–299. IEEE Computer Society, 2009.

[50] H. Sariowan, R. Cruz, e G. Polyzos. Scheduling for quality of service guarantees via service curves. *In Proceedings of the International Conference on Computer Communications and Networks*, páginas 512–524, 1995.

[51] P. Valente e F. Checconi. BFQ and related stuff on disk scheduling. `http://algo.ing.unimo.it/people/paolo/disk_sched/`. Acessado em 14 de Abril, 2011.

[52] Intel Corporation and Seagate Technology. Native Command Queuing, an exciting new performance feature for Serial ATA. `http://www.seagate.com/content/pdf/`

whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf. Acessado em 20 de Maio, 2011.

PEDRO EUGÊNIO ROCHA

SISTEMAS DE ARMAZENAMENTO COMPARTILHADO COM QUALIDADE DE SERVIÇO E ALTO DESEMPENHO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Luis Carlos Erpen de Bona

CURITIBA

2011