Um Simulador para a Arquitetura RAID5

Hermano Peixoto de Oliveira Júnior

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DO IMECC DA UNICAMP

Oliveira Junior, Hermano Peixoto

O41u Um simulador para a arquitetura RAID5 / Hermano Peixoto de Oliveira Junior. -- Campinas, [SP : s.n.], 1993.

Orientador : Celio Cardoso Guimarães.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática [illegible].

1. Discos magnéticos. 2. MS-DOS (Sistema operacional de computador). I. Guimarães, Celio Cardoso. II. Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Ciência da Computação. III. Título.

Um Simulador para a Arquitetura RAID5

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Hermano Peixoto de Oliveira Júnior e aprovada pela Comissão Julgadora.

Campinas, 6 de dezembro de 1993.

Prof. Célio Cardoso Guimarães
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtensão do título de Mestre em Ciência da Computação.

Um Simulador para a Arquitetura RAID5[1]

Hermano Peixoto de Oliveira Júnior[2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- Ricardo Anido (Suplente)[3]
- Célio Cardoso Guimarães (Orientador)[3]
- Nélson Castro Machado (Coorientador)[3]
- Márcio Luís de Andrade Netto[4]

[1]Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtensão do título de Mestre em Ciência da Computação.

[2]O autor é Bacharel em Engenharia Elétrica pela Universidade Federal da Bahia.

[3]Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

[4]Professor do Departamento de Engenharia de Computação e Automação Industrial - FEE - UNICAMP.

A meus pais, a meus irmãos, e ...

... a Cecilia.

Agradecimentos

Ao CNPq pelo apoio financeiro recebido.

Aos meus orientadores, Célio e Machado, pela confiança, apoio e amizade que caracterizaram esses últimos três anos.

A Jorge Omar, por suas sugestões na fase de definição deste trabalho.

A meus grandes amigos Hernán e Fátima, pela amizade e pelo apoio que sempre recebi.

A Maria, Antônio, João Victor e Florência, pelo entrosamento que temos e pelas horas divertidas que passamos juntos.

A Maria do Lago e a todos de sua família que me receberam em Campinas como se eu fosse mais um membro da família.

A José Lino e a Marcelo Andrade por sua valiosa amizade.

Aos amigos da Autitec, pelo apoio recebido neste ano.

Por último, gostaria de agradecer a todos os amigos que fiz na Unicamp. Em especial aos amigos do Volley.

Resumo

A performance dos dispositivos de I/O em um sistema de computação não tem acompanhado o desenvolvimento da unidade central de processamento (CPU). Como resultado, o poder computacional das máquinas que fazem uso de uma grande quantidade de I/O tem sido desperdiçado. Como exemplo, a performance de um servidor de arquivos é severamente limitada pela performance do disco magnético. Esta tese se concentra neste dispositivo de I/O. Um simulador de um subsistema de discos magnéticos, baseado na arquitetura RAID 5 proposta por Patterson, é apresentado.

Abstract

The performance of I/O devices in a computing system has not been following the developments of the Central Processing Unit (CPU). As a result, the computational power of machines which make use of a large amount of I/O has been largely worthless. As an example, the performance of a file server is severely limited by the performance of the magnetic disc. This thesis is focused on this I/O device. A simulator of a magnetic disc subsystem, based on the RAID 5 architecture proposed by Patterson, is presented here.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este capítulo apresenta, de forma sucinta, um sumário do problema abordado, a motivação para a construção do simulador RAID 5, os objetivos a serem alcançados e a arquitetura do simulador. Por questões de clareza, incluímos uma visão geral sobre a arquitetura RAID 5, apresentando seus pontos principais.

1.1 Introdução

A capacidade de processamento dos computadores vem crescendo rapidamente. Contudo, uma CPU rápida não necessariamente torna um sistema rápido. A velocidade com a qual as instruções e dados são fornecidas à CPU também determina seu desempenho. Por exemplo, considere uma estação de trabalho RISC capaz de executar $50 MIPS$ (*millions of instructions per second*). Teoricamente, esta estação poderia acessar posições de memória e periféricos a cada $20ns$. Contudo, periféricos muito lentos tais como aqueles diretamente manipulados pelo homem, são a causa de um enorme desperdício de poder de processamento da CPU. Inclusive, esta é uma das razões para que os modernos sistemas operacionais suportem múltiplas tarefas. Enquanto uma tarefa estiver aguardando por um periférico lento, a CPU pode executar outra tarefa, aumentando o desempenho do sistema. Citando o teclado como exemplo, temos que um bom digitador não consegue entrar com mais de 10 teclas por segundo, ou seja, mais que uma tecla a cada $100ms$. Outra classe importante de periférico é aquela dos não diretamente manipulados pelo homem, tais como impressoras, traçadores gráficos, discos magnéticos rígidos (*hard disks*), etc. Esse tipo de dispositivo também limita o desempenho da CPU. Neste caso, é a inércia mecânica a responsável por este fato. Citando os discos magnéticos como exemplo, temos que a taxa de transferência de dados da maioria desses dispositivos é inferior a $1.5 MB/s$ (1 *byte* a cada $667ns$), ou seja, 34 vezes menor que a capacidade nominal da CPU. Como esse dispositivo é muito usado pelo mecanismo de memória virtual e pelo sistema de arquivos, seu impacto no desempenho da máquina é muito grande. Existe, entretanto, uma diferença fundamental entre as duas classes citadas de dispositivos de I/O. A primeira dificilmente pode ser melhorada em relação a velocidade. O operador humano é um fator limitante muito importante.

Para suavizar os efeitos do baixo desempenho dos dispositivos de I/O não diretamente manipulados pelo homem, costuma-se usar *bufferização* por *software*. Contudo, inovações tecnológicas são

necessárias, como demonstrado pela Lei de Amdahl [Amd 67]:

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

onde

- S = *speedup* efetivo;
- f = fração do trabalho no modo mais rápido;
- k = *speedup* no modo mais rápido.

Por exemplo, suponha que alguma aplicação utilize 10% do seu tempo em transações com um periférico lento. Quando as CPUs forem dez vezes mais rápidas o *speedup* efetivo será de apenas cinco vezes. Quando tivermos CPUs cem vezes mais rápidas, essa aplicação rodará menos de dez vezes mais rapidamente, desperdiçando 90% do *speedup* potencial. Esse exemplo mostra que é muito necessário o estudo de melhorias de *hardware* para os diversos tipos de periféricos.

1.2 Escopo da Dissertação

Nesta dissertação, abordamos o problema da melhoria de desempenho de um dispositivo de I/O em particular: o disco magnético rígido. Apresentamos algumas propostas de arquitetura para subsistemas de discos magnéticos, nos concentramos na proposta RAID 5 [Pat 88] e apresentamos um simulador para esta arquitetura.

1.3 Objetivos da Dissertação

Motivados pelo problema da redução do desempenho de um sistema de computador devido ao baixo desempenho dos discos magnéticos quando comparada ao desempenho da CPU, decidimos construir um simulador, em ambiente MSDOS, para uma nova proposta de arquitetura de subsistema de discos magnéticos (RAID 5), de modo a satisfazer o seguinte requisito.

A maior parte do seu código fonte deve poder ser utilizado, sem modificação de qualquer espécie, em uma controladora RAID 5 que venha a ser construída.

Esta é a principal contribuição desta dissertação. Este requisito suprime do implementador do *hardware* de controle a tarefa de codificar os algoritmos de posicionamento dos dados e informação de redundância, relativos a essa arquitetura. Além disso, ele fica livre da parte de depuração dos algoritmos RAID 5. É muito difícil depurar um *software* dedicado a controle pois, quase sempre, não há interface de usuário adequada. Para nós, implementadores dos algoritmos RAID 5, o fato de termos utilizado o MSDOS como plataforma de simulação permitiu o desenvolvimento e, principalmente, a depuração dos algoritmos em um ambiente muito mais amigável do que aquele de um *hardware* de controle. Isto refletiu no aumento da confiabilidade da codificação dos algoritmos. No ambiente MSDOS, é muito fácil detectar e corrigir erros em programas.

Decidimos não usar o ambiente UNIX devido a falta de disponibilidade de máquina. Teríamos que dedicar uma máquina ao projeto, pois, como estaríamos operando a nível do núcleo do sistema operacional, poderíamos causar um colapso no sistema e interferir na operação dos demais usuários da máquina.

Como optamos pelo ambiente MSDOS, o seguinte requisito também teve que ser levado em consideração.

o disco RAID 5 implementado deve ser indistinguível, do ponto de vista dos aplicativos e dos usuários, de qualquer disco físico existente no sistema; Isto para evitar que programas de teste especiais tenham que ser desenvolvidos.

Na verdade, este requisito é apenas uma comodidade visando as fases de teste e depuração. Não tem relação direta com o objetivo principal deste trabalho.

Aqui, não estamos interessados em medições de desempenho. Temos duas razões para isto. Primeiro, essas medições já foram extensivamente feitas [Gib 89]. Segundo, o MSDOS não é um ambiente adequado para tais medições. Isto devido às suas limitações de desempenho. Nosso objetivo é gerar a parte fundamental do *software* de um *hardware* de controle RAID 5.

A escolha da arquitetura RAID 5 foi fundamentada no fato de que suas características técnicas são as que mais se adequam à maioria das aplicações reais: as de processamento de transações e as de supercomputadores.

1.4 A Arquitetura do Simulador RAID 5

Nesta seção, apresentamos de forma genérica a arquitetura RAID 5 e a estrutura geral do simulador. Esses temas são discutidos em detalhes nos Capítulos 4 e 5, respectivamente.

1.4.1 A Arquitetura RAID 5

Matrizes de disco têm sido tradicionalmente usadas em supercomputadores para se obter altas taxas de transferência de dados. Isto é conseguido através de entrelaçamento de dados e acessos simultâneos aos discos componentes da matriz [Kim 86]. Ou seja, ao invés de se obter x MB/s (*Mega Bytes* por segundo) de um único disco, obtemos uma taxa de transferência dessa ordem de grandeza mediante acesso simultâneo a N discos com taxa de transferência individual de (x/N) MB/s. Não obtemos exatamente x MB/s devido às informações de redundância que têm que ser adicionadas à matriz, a fim de se manter a integridade dos dados.

RAID 5 dedica o equivalente a um disco para o armazenamento da informação de redundância. Contudo, a distribui entre todos os discos da matriz. O Capítulo 4 apresenta os diversos tipos de mapeamento dessa informação. Como ilustração, podemos citar o mapeamento assimétrico esquerdo, mostrado na Figura 1.1.

Embora definamos precisamente faixa de redundância no Capítulo 4, por hora, simplifiquemos o conceito, dizendo que a i-ésima faixa de redundância é a reunião dos i-ésimo setores de todos os discos na matriz. Assim, no mapeamento assimétrico esquerdo, os dados são colocados seqüencialmente, da esquerda para a direita, nas faixas de redundância. Para cada faixa sucessiva,

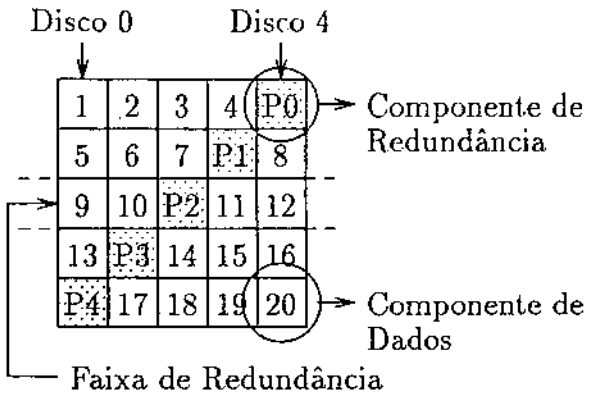

Figura 1.1: Mapeamento Assimétrico-Esquerdo

o ponto no qual a informação de redundância é inserida é rotacionado em uma unidade para a esquerda.

O posicionamento da redundância em RAID 5 possibilita a ocorrência de múltiplas escritas em paralelo. Por exemplo, como na Figura 1.2, suponhamos que temos cinco discos em um grupo e desejamos escrever no setor 1 do disco 1 (B) e no setor 2 do disco 2 (C). Essas escritas podem ocorrer em paralelo pois uma escrita em B apenas envolve uma escrita no disco 4 enquanto uma escrita em C apenas envolve uma escrita no disco 0.

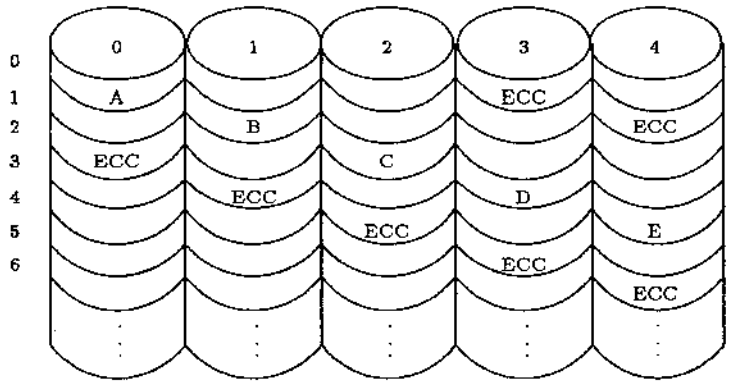

Figura 1.2: Neste exemplo, a informação de ECC do setor 0 está no disco 3, a do setor 1 está no disco 2 e a do setor 2 está no disco 0. Assim, os segmentos de dados B e C podem ser escritos simultaneamente. O mesmo não acontece com os segmentos A e C.

Em qualquer instante, a informação de redundância presente na matriz RAID 5 deve ser capaz de propiciar confiabilidade, isto é, deve poder prover meios de recuperação de erros. Isto significa que a cada operação de escrita na matriz, esta informação deve ser atualizada, de modo que esta

premissa continue válida.

O cálculo é muito simples. Faz-se um XOR de todas as componentes da faixa de redundância que deverão ser atualizadas na matriz, juntamente com a componente de redundância associada. Ou seja, se supusermos (sem perda de generalidade) que S_1, S_2, ..., S_n é um subconjunto a ser atualizado de uma determinada faixa de redundância cuja informação de redundância atual seja R, então a nova informação de redundância é dada por:

$$R = R_{anterior} \bigoplus S_1 \bigoplus S_2 \bigoplus \cdots \bigoplus S_n \bigoplus S_{1,anterior} \bigoplus S_{2,anterior} \bigoplus \cdots \bigoplus S_{n,anterior}.$$

O cálculo da informação de redundância, como descrito no parágrafo anterior, faz com que o método de recuperação de erros seja simples. Quando uma componente da faixa de redundância lida for considerada defeituosa, esta pode ser reconstruída computando-se o XOR das componentes restantes, juntamente com o conteúdo da componente de redundância. Ou seja, se a componente S_k de uma faixa de redundância com n discos estiver defeituosa, podemos recuperá-la através da expressão

$$S_k = S_1 \bigoplus \cdots \bigoplus S_{k-1} \bigoplus S_{k+1} \bigoplus \cdots \bigoplus S_n \bigoplus R.$$

1.4.2 O Simulador RAID 5

Como o ambiente da simulação RAID 5 é o MSDOS, um dos nossos requisitos de projeto requer que o disco RAID 5 seja indistinguível de qualquer outro disco presente no sistema. Em MSDOS, a única solução para este requisito é implementar o simulador como sendo um *device driver* de bloco[1].

O requisito principal, desejo de que a maioria do seu código possa ser empregado, sem alteração de qualquer espécie, em um *hardware* de controle dedicado, determinou que o simulador fosse composto de três módulos independentes, como mostrado na Figura 1.3. O primeiro, implementa todo o código intrínseco a um *device driver* de bloco. Faz a interface entre o núcleo do MSDOS e o núcleo do simulador. Este segundo módulo não sabe que está sendo executado num contexto de *device driver*, nem que tipo de dispositivo de armazenamento secundário está de fato comandando. O terceiro módulo é o responsável por executar o mapeamento dos discos lógicos, vistos pelo núcleo do simulador, em entidades físicas. Em nosso caso, arquivos do MSDOS. O mecanismo de comunicação entre os diversos módulos do simulador é a troca de mensagens.

Como mapeamos os discos lógicos da matriz RAID 5 em arquivos MSDOS, nada mais natural que pedir ao próprio MSDOS que execute as operações de escrita e leitura nesses arquivos. Essa idéia, embora aparentemente simples, é de difícil implementação pois, se assim for feito, o MSDOS deverá ser reentrado. Como sabemos, todas as referências sobre MSDOS afirmam que ele não é reentrante e que temos que conviver com este fato. Contudo, o conhecimento da estrutura interna do MSDOS nos permitiu implementar essa característica.

Muito trabalho foi despendido implementando-se a reentrância ao MSDOS. Contudo, o código que implementa essa possibilidade pode ser isolado e utilizado em outros projetos.

1.5 Organização da Dissertação

O Capítulo 2 (*Sumário da Evolução dos Discos Magnéticos*) apresenta um sumário do desenvolvimento tecnológico dos discos magnéticos, desde o primeiro construído até os atuais, em quatro

[1] Vide Apêndice A

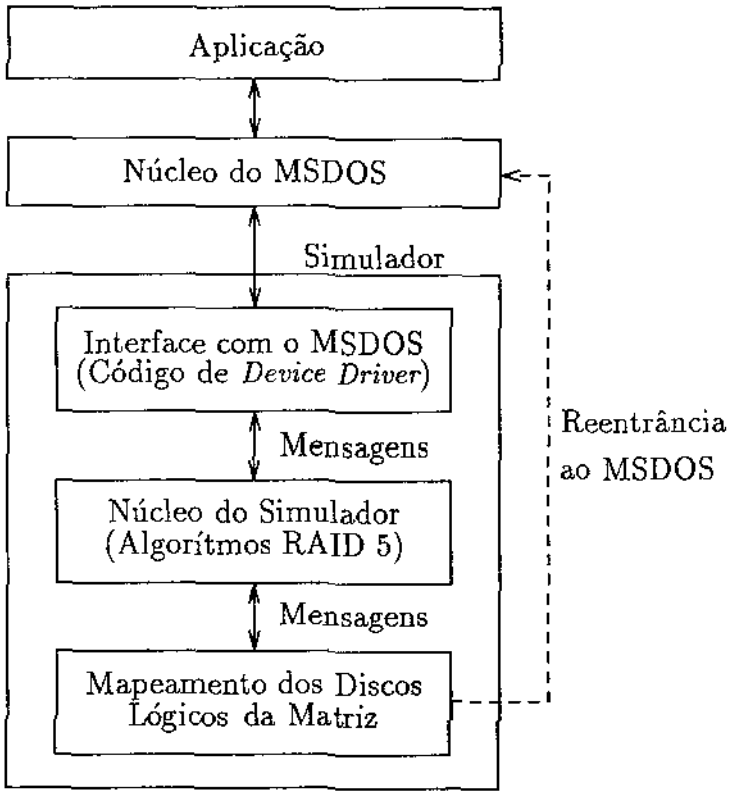

Figura 1.3: Arquitetura do Simulador RAID 5

áreas fundamentais: a cabeça magnética e sua sustentação; O atuador de posicionamento da cabeça magnética; O substrato do disco e sua cobertura magnética; A codificação dos dados e a eletrônica de leitura/gravação. Não é discutida a aplicação desses dispositivos em subsistemas de armazenamento secundário.

O Capítulo 3 (*Arquiteturas de Alto Desempenho para Subsistemas de Disco*) apresenta, resumidamente, técnicas de construção de subsistemas de disco que permitem reduzir o tempo de posicionamento, a latência rotacional e o tempo de transferência. Discutimos as considerações que levaram ao interesse pelo problema, e apresentamos duas classes de abordagem para o problema: extensões a arquiteturas convencionais e novas propostas de arquiteturas.

O Capítulo 4 (*A Arquitetura RAID 5*) apresenta em detalhes a arquitetura RAID 5. Mostra as diversas formas de mapemento lógico para físico de um disco RAID 5, o cálculo da redundância e recuperação de erros, a confiabilidade e o desempenho desta arquitetura.

O Capítulo 5 (*Um Simulador Para a Arquitetura RAID 5*) apresenta um simulador para a arquitetura RAID 5 para as versões 3.1 a 6.0 do sistema operacional MSDOS, implementado como *device driver* de bloco. Apresenta também um histórico das decisões de projeto e um histórico do desenvolvimento da implementação.

O Capítulo 6 (*Considerações Finais e Trabalhos Futuros*) apresenta as conclusões deste trabalho, seguidas de algumas sugestões visando futuros estudos nas áreas abrangidas por este projeto.

O Apêndice A (*Uma Visão Geral de Algumas Estruturas do MSDOS*) apresenta as principais estruturas internas e externas do MSDOS, documentadas e não documentadas pela *Microsoft*, que foram utilizadas na implementação do simulador RAID 5, apresentado no Capítulo 5.

Capítulo 2

Sumário da Evolução dos Discos Magnéticos

Neste capítulo, apresentamos um sumário do desenvolvimento tecnológico dos discos magnéticos, desde o primeiro construído até os atuais, em quatro áreas fundamentais: a cabeça magnética e sua sustentação; o atuador de posicionamento da cabeça magnética; o substrato do disco e sua cobertura magnética; a codificação dos dados e a eletrônica de leitura/gravação. Não discutiremos a aplicação desses dispositivos em subsistemas de armazenamento secundário.

A redação desse capítulo foi fortemente influenciada pela referência [Har 81], onde se encontra um bom histórico da evolução dos discos magnéticos entre 1957 e 1981.

2.1 Introdução

O primeiro disco magnético (IBM 350) foi desenvolvido por M. L. Lesser, J. W. Haanstra, T. Noyes e W. E. Dickinson no final da década de 50 e apresentado à comunidade científica através dos artigos [Les 57] e [Noy 57]. Suas características principais são mostradas na Figura 2.1. Três delas influem diretamente no desempenho em operação:

1. tempo médio de posicionamento (*average seek time*): tempo necessário para se deslocar entre duas trilhas randomicamente selecionadas;

2. latência rotacional média (*average rotational latency*): tempo necessário para posicionar as cabeças no setor desejado, já estando sobre a trilha correta; Aproximadamente, metade do tempo de revolução do disco;

3. tempo médio de transferência de dados (*average data transfer time*): tempo requerido para transferir os *bytes* do disco para o *Host*.

As justificativas para a utilização de um meio magnético como base da tecnologia dos discos foram a não volatilidade, a recuperação imediata dos dados armazenados, a reversibilidade ilimitada e a relativa simplicidade e baixo custo dos transdutores e do meio magnético.

Em 1981, quase 25 anos após o lançamento do IBM 350, foi lançado o modelo IBM 3380, apresentando evoluções tecnológicas nem sequer imaginadas na época do IBM 350. Suas características principais são mostradas na Figura 2.2.

Ano de Lançamento	1957
Densidade de Gravação	
Densidade de Área (Mb/in^2)	0.002
Densidade Linear (bpi)	100
Densidade de Trilhas (tpi)	20
Parâmetros Geométricos (μin)	
Espaçamento Cabeça-Disco	800
Comprimento do Entreferro	1000
Espessura do Meio	1200
Suspensão e Elemento Magnético	
Tipo da Suspensão	Hidrostática
Contorno da Superfície	Plana
Material do Atuador	Al
Material do Núcleo	Metal
Conexão Atuador/Núcleo	Epoxy

Disco	
Diâmetro (in)	24
Espessura do Substrato (in)	0.1
RPM	1200
Superfícies/Rotor	100
Atuador	
Geometria de Acesso	x-y
Cabeças	2 por Atuador
Atuadores/Rotor	3
Tempo Médio de Posicion. (ms)	600
Eletrônica de Leitura/Gravação	
Taxa de Dados ($KBytes/s$)	8.8
Codificação	NRZI
Detecção	Amplitude
Referência	2 Osciladores

Figura 2.1: Características do Disco IBM 350

Ano de Lançamento	1981
Densidade de Gravação	
Densidade de Área (Mb/in^2)	> 12
Densidade Linear (bpi)	15200
Densidade de Trilhas (tpi)	> 800
Parâmetros Geométricos (μin)	
Espaçamento Cabeça-Disco	< 13
Comprimento do Entreferro	25
Espessura do Meio	< 25
Suspensão e Elemento Magnético	
Tipo da Suspensão	Hidrodinâmica
Contorno da Superfície	Ponta Plana
Material do Atuador	Cerâmica
Material do Núcleo	Filme
Conexão Atuador/Núcleo	Deposição

Disco	
Diâmetro (in)	14
Espessura do Substrato (in)	> 0.075
RPM	3620
Superfícies/Rotor	15
Atuador	
Geometria de Acesso	Linear
Cabeças	2 por Superfície
Atuadores/Rotor	2
Tempo Médio de Posicion. (ms)	16
Eletrônica de Leitura/Gravação	
Taxa de Dados ($KBytes/s$)	3000
Codificação	RLL 2, 7
Detecção	Delta Clip
Referência	VFO

Figura 2.2: Características do Disco IBM 3380

Os avanços tecnológicos que permitiram tamanha disparidade de desempenho serão discutidos, sob quatro pontos de vista interrelacionados:

1. a cabeça magnética e sua sustentação; É fundamental que o espaçamento entre a cabeça magnética e a superfície do disco seja o menor possível, para faciltar a gravação em alta densidade;

2. o desenvolvimento mecânico do atuador de posicionamento da cabeça magnética em trilhas concêntricas de um disco em rotação;

3. o substrato do disco e sua cobertura magnética;

4. a eletrônica requerida para ler e escrever, confiavelmente, dados no disco.

2.2 A Cabeça Magnética e Sua Sustentação

Ao longo da história dos discos, dois tipos de sustentação foram empregados para manter a cabeça magnética afastada do disco: sustentação hidrostática e sustentação hidrodinâmica. Ela é necessária para prevenir desgastes tanto da cabeça como do meio magnético.

2.2.1 Sustentação Hidrostática

Em junho de 1953, W. A. Goddard [God 65] demonstrou a operação de uma cabeça magnética com sustentação hidrostática alimentada por ar comprimido. Ela possuía pequenos furos em sua face, ao longo de uma circunferência, de modo que o fluxo de ar que saia dela, em direção à superfície magnética, provia a força necessária para contrabalançar a suspensão realizada por uma mola. O transdutor montado no centro escreveu e leu, com sucesso, padrões de *bits*, estabelecendo a viabilidade de se manter um espaçamento consistente entre a superfície do disco e a cabeça magnética.

No mesmo ano, N. A. Voegel [Voe 55] aperfeiçoou a técnica de Goddard. Ele mostrou que, para manter a estabilidade, era necessário haver um orifício para escape de ar no centro da cabeça magnética, a qual era aproximada da superfície do disco por três pistões conectados à mesma câmara de ar que supria os furos da superfície de sustentação. Aplicada a pressão, a cabeça era forçada contra o disco com uma carga proporcional à sustentação, até que a distância atingisse $800\mu in$. Pequenas molas retornavam a cabeça à sua posição inicial, quando a pressão era removida.

A diferença de desempenho entre esses dois projetos vem do fato de que, no de Goddard, o fluxo de ar afasta a cabeça da superfície do disco, e no de Voegel, aproxima. Isso porque, aproximando, efeitos aerodinâmicos, auxiliados pelo furo de escape, facilitam manter a constância do espaçamento entre a cabeça e a superfície.

2.2.2 Sustentação Hidrodinâmica

J. J. Hagopian concebeu o uso de sustentação hidrodinâmica, mostrando que o filme de ar que aparece entre a cabeça magnética e a superfície do disco em rotação poderia gerar uma sustentação razoavelmente plana. Baseado nesse conceito, um novo projeto foi iniciado. Com seu prosseguimento, ficou aparente que a operação dessa sustentação era errática, pois se desconsiderava o efeito da compressibilidade do ar. W. A. Gross [Gro 59], R. K. Brunner, J. M. Harker, K. E. Haughton

e A. G. Osterlund [Bru 59] estudaram esse problema e concluíram que uma cavidade deveria ser provida para a entrada da camada limite do filme de ar. Teoricamente, essa sustentação operaria com sucesso em superfícies planas, mas haveriam problemas de instabilidade nas superfícies reais. Brunner e Haughton mostraram que a curvatura local da superfície do disco era da mesma ordem de magnitude que a convexidade do deslizador da cabeça[1], podendo produzir regiões de instabilidade durante a rotação. Eles também mostraram que a rotação do disco poderia produzir acelerações locais que causariam excessiva mudança de espaçamento.

Esses trabalhos tornaram claro que a especificação de planaridade do disco, em termos da curvatura local e da velocidade e aceleração da rotação, era fundamental no projeto unidades de disco mais sofisticadas.

2.3 O Atuador de Posicionamento da Cabeça Magnética

A tarefa do mecanismo de posicionamento é transportar a cabeça magnética de uma trilha para outra, de forma rápida e precisa. A velocidade é importante porque afeta diretamente o tempo de resposta do sistema e a precisão porque influi na densidade máxima de trilhas.

Apresentaremos, a seguir, os atuadores que tiveram mais influência na evolução da tecnologia dos discos magnéticos.

2.3.1 O Atuador do IBM 350

O IBM 350 possuía cinqüenta superfícies magnéticas, de $24in$ de diâmetro cada, ligadas a um mesmo eixo de rotação. Haviam duas cabeças magnéticas montadas numa estrutura que era movida verticalmente a um dos 50 discos por um cabo conectado a um motor. Usava-se um servo para posicionar a estrutura no disco selecionado e outro para mover as cabeças para uma das 100 trilhas existentes. Ao se chegar à trilha estabelecida, ar comprimido era forçado contra a superfície magnética para permitir a sustentação das cabeças. O tempo de posicionamento deste atuador era $600ms$, em média, e a densidade de trilhas $20tpi$.

2.3.2 O Atuador do IBM 1301

O disco IBM 1301, lançado em 1962, foi um sucessor direto do IBM 350. As principais modificações foram a adoção de sustentação hidrodinâmica e o mecanismo de posicionamento. Neste modelo, existia um atuador para cada superfície magnética, que empregava dois conjuntos de cilindros hidráulicos para posicionamento grosso e fino, cuja precisão permitiu uma densidade de trilhas de $100tpi$. O tempo médio de posicionamento ficou em $165ms$.

Este foi o primeiro modelo no qual se definiu o conceito de *cilindro*. O i-ésimo cilindro é a reunião de todas as i-ésimas trilhas de todas as superfícies magnéticas.

2.3.3 O Atuador do IBM 1311

O disco IBM 1311, lançado em 1963, marcou o início da era das superfícies magnéticas removíveis. Existiam módulos, trocáveis pelos usuários, com seis superfícies de $14in$ de diâmetro encerradas numa proteção. O processo era simples. Removia-se o módulo corrente após o encaixe da proteção,

[1] elemento mecânico responsável pelo deslocamento da cabeça entre trilhas

inseria-se o novo módulo e retirava-se sua proteção. O sistema mecânico de encaixe era tal que só permitia inserção e remoção quando o módulo estava posicionado no encaixe do disco.

O atuador do IBM 1311 movia cada cabeça (uma por superfície) em duas velocidades. A maior era usada para mover por longas distâncias e a menor selecionada logo antes de se chegar ao cilindro desejado. Esta combinação resultou num atuador de baixo custo, com tempo de acesso de $150ms$. Esse conceito foi utilizado também nos discos posteriores, modelos IBM 2311 e IBM 2314. Nesses, a troca do atuador de duas velocidades para um de três reduziu o tempo de posicionamento para $60ms$.

2.3.4 O Atuador do IBM 3330

O IBM 3330, lançado em 1971, também possuía superfícies removíveis. Sua principal inovação foi incorporar um mecanismo de realimentação no posicionamento da cabeça, constituído de:

- uma superfície com trilhas especiais pré-gravadas que definiam o posicionamento de cada cilindro;
- uma servo-cabeça para ler tais trilhas;
- um sistema de controle para gerar um sinal de erro de posicionamento para acionar o atuador;
- um motor para o atuador que gerava uma força proporcional ao sinal de erro.

A movimentação das cabeças até o cilindro desejado, era feito por uma bobina cilíndrica que se movia num campo magnético similar ao de um auto-falante (*voice coil motor*). Durante o posicionamento, a corrente na bobina era controlada, monitorando-se o cruzamento de trilhas. O padrão gravado nas servo-trilhas foi um fator chave na precisão do posicionamento a um custo razoável. Este tipo de atuador resultou num tempo de posicionamento de $30ms$.

2.3.5 O Atuador do IBM 3340

O IBM 3340, lançado em 1973, continha muitas inovações. A principal foi o uso de uma técnica que ficou conhecida como *Winchester*, onde as cabeças magnéticas possuíam massa tão pequena que a lubrificação dos discos as permitiam permanecer em contato com a superfície magnética enquanto o posicionamento iniciava e finalizava, eliminando o mecanismo de levantar e baixar a cabeça. Atingiu-se uma capacidade de $70MBytes$ através de quatro superfícies, cada qual com duas cabeças, com densidade de trilhas de $300tpi$ e tempo de posicionamento $25ms$.

2.3.6 O Atuador do IBM System/32

O atuador que mais influenciou o mercado atual de discos magnéticos foi desenvolvido no IBM Development Laboratory em Hursley, Inglaterra. Era do tipo rotativo com apenas uma parte móvel, provendo as vantagens do atuador *voice coil* a um custo bem mais baixo. Este atuador, combinado com a tecnologia *Winchester* em um invólucro selado, foi utilizado por diversos sistemas tais como o IBM System/32. O calor gerado por esse pequeno dispositivo era pequeno o suficiente para ser dissipado pelo próprio invólucro, sem necessidade de refrigeração forçada.

2.4 O Substrato do Disco e Sua Cobertura Magnética

2.4.1 Substrato

O primeiro disco magnético (IBM 305) utilizou um substrato de alumínio laminado, com imperfeições relativamente acentuadas. A solução para esse problema foi utilizar uma cobertura magnética espessa, da ordem de $1200\mu in$. Esta técnica se mostrou inviável para para gravação em maior densidade, onde se requer que a cabeça esteja bem próxima da superfície. Assim, várias técnicas de polimento foram desenvolvidas para melhorar a planaridade e a suavidade da superfície dos discos.

Com a introdução da tecnologia *Winchester* e o aumento da velocidade de rotação, as freqüências de ressonância do substrato se tornaram importantes. Assim, a velocidade de rotação e a espessura do substrato eram escolhidas de modo a evitar essas freqüências.

2.4.2 Cobertura Magnética

A cobertura era formada pela dispersão de partículas magnéticas no substrato. No início dos anos 50, essas partículas estavam limitadas à gama forma do óxido de ferro. A indústria de polímeros estava em sua infância e provia poucas opções de material que satisfizessem as rigorosas especificações do meio magnético.

As propriedades magnéticas das partículas disponíveis comercialmente continuaram a melhorar com o passar dos anos. Bom controle sobre sua geometria produziu materiais que facilitaram a orientação no meio de gravação. Defeitos estruturais diminuíram, minimizando os poros e inclusões não magnéticas entre as partículas.

Orientação das Partículas Magnéticas

A orientação das partículas do óxido, como rotineiramente praticado no caso das fitas magnéticas, tinha sido, por muito tempo, objeto de estudo. E. M. Williams descobriu que uma nova partícula magnética da Pfizer tinha características que permitiam a orientação de modo fácil. Este óxido levou a um novo produto com melhorias substanciais nas propriedades de gravação, o IBM 3340.

Os parâmetros chave para a orientação são a viscosidade do meio, o tempo requerido para orientar as partículas, e o campo magnético aplicado. Devido à orientação vir do campo de um entreferro, a intensidade e o tempo que as partículas vêm o campo são fatores críticos para a boa orientação. Usando um par de magnetos opostos, em cada lado do disco, de modo que o campo seja essencialmente no plano do disco, foi achado que o campo do entreferro deveria estar na faixa de 1800 a $2000 Oersteds$. Se fosse menor que isto, existiria uma força insuficiente para girar as partículas num meio viscos. Se fosse muito maior, as partículas simplesmente mudariam sua polaridade ao invés de serem giradas. Era também necessário girar o disco devagar para a máxima orientação. Se este fosse girado muito depressa, as partículas trocariam sua polarização magnética, e se muito devagar, causaria imperfeições na cobertura magnética.

Lubrificação

O desenvolvimento da tecnologia *Winchester* gerou novos requerimentos nas características da superfície de gravação, sendo satisfeitos por técnicas de lubrificação similares às das fitas magnéticas. O primeiro lubrificante usado foi uma solução a 0.75% do óleo RDC-200 em freon. Nos anos

subseqüentes, muitos outros lubrificantes foram avaliados para atender os requisitos dos discos mais modernos.

2.5 Codificação dos Dados e a Eletrônica de Leitura/Gravação

O primeiro disco utilizou codificação NRZI, na qual os dados eram gravados mudando-se a orientação do meio magnético nos *bits* 1 e mantendo-se nos *bits* 0. Na leitura, um pulso era gerado para cada detecção de um *bit* 1. A presença do sinal era detectado quando a tensão elétrica na cabeça excedia um nível predeterminado. Os circuitos de sincronização utilizavam dois osciladores de partida rápida, cada qual iniciado por *bits* 1 alternados. Após a partida, os ciclos de um oscilador eram usados para referenciar todos os *bits* 0 subseqüentes. O segundo oscilador era iniciado quando o próximo *bit* 1 fosse encontrado.

Esse método sofria da dificuldade em se produzir uma eletrônica confiável para o circuito de sincronização. O oscilador de freqüência variável (*variable-frequency oscilator*), VFO, foi um circuito que ofereceu o potencial para reduzir a maioria das tolerâncias experimentadas pelos circuitos anteriores. O circuito era um oscilador cuja freqüência podia ser ajustada eletronicamente via uma malha de realimentação.

A precisão do VFO em altas freqüências permitiu a utilização de um método de codificação com menos transições por *bit*, cuja estrutura do código o tornava susceptível a ruído. Assim, um novo sistema de detecção conhecido como delta-V foi desenvolvido. Este detector empregava uma série de testes para melhorar a rejeição ao ruído: uma inclinação de subida mínima era requerida; o pico do sinal era detectado achando-se a derivada do ponto de *crossover*; uma inclinação de descida mínima era sentida; a amplitude do sinal associado ao pulso tinha que exceder um nível mínimo. Através de uma linha de retardo, essas quatro condições eram testadas e, se satisfeitas, um pulso de saída era gerado.

Mais recentemente, códigos *run-length-limited* foram implementados. Esses códigos limitam a mínima e a máxima distância entre as transições. Por exemplo, o código 2,7 tem um mínimo de 2 e um máximo de 7 zeros entre as transições. Ao mesmo tempo, eles usam as transições com tal eficiência que, em média, 1.5 transições por *bit* são conseguidas. O sinal é testado para ver se a polaridade dos pulsos alternam e se têm uma amplitude mínima. O pico do sinal é obtido através de diferenciação e, finalmente, uma mudança mínima de sinal, após o pico, deve ocorrer dentro de um período específico de tempo. Esses quatro critérios são usados para se tomar uma decisão na detecção.

Capítulo 3

Arquiteturas de Alto Desempenho para Subsistemas de Disco

Neste capítulo, apresentamos técnicas de construção de subsistemas de disco que permitem reduzir o tempo de posicionamento, a latência rotacional e o tempo de transferência. Inicialmente, discutimos as considerações que levaram ao interesse pelo tema, e, a seguir, apresentamos duas classes de abordagem para o problema:

1. extensões a arquiteturas convencionais;
2. novas propostas de arquitetura.

No escopo da segunda classe, estão as arquiteturas RAID. Dentre estas, apresentaremos em detalhes, no próximo capítulo, a arquitetura RAID 5, objeto desta tese.

3.1 Introdução

A grande disparidade entre a velocidade de processamento da CPU de um computador e a velocidade de seus dispositivos de I/O tem consumido muito tempo de pesquisa por parte de fabricantes e pesquisadores universitários, pois tais dispositivos limitam severamente o desempenho do sistema. Várias técnicas, algumas das quais serão vistas neste capítulo, foram e estão sendo desenvolvidas no intuito de resolver esse problema.

Durante a última década, os microprocessadores experimentaram um crescimento na capacidade de processamento bem maior que as outras classes de uniprocessadores. Bell [Bel 84] observou que o desempenho dos microprocessadores aumentou a uma taxa de 40% ao ano entre 1974 e 1984, aproximadamente o dobro da taxa dos minicomputadores. Já, Myers [Mye 86], utilizando a família Intel, observou que a capacidade de processamento dos microprocessadores dobrou a cada 2.25 anos desde 1978. Outro pesquisador, Joy, predisse uma taxa de crescimento ainda maior:

$$MIPS = 2^{ano-1984}.$$

Podemos ver, na Figura 3.1, o rápido crescimento do desempenho dos microprocessadores como observado por Myers e predito por Joy.

Contudo, uma CPU rápida não necessariamente torna um sistema rápido. O subsistema de memória deve também se tornar mais rápido e maior para se adequar ao aumento na demanda

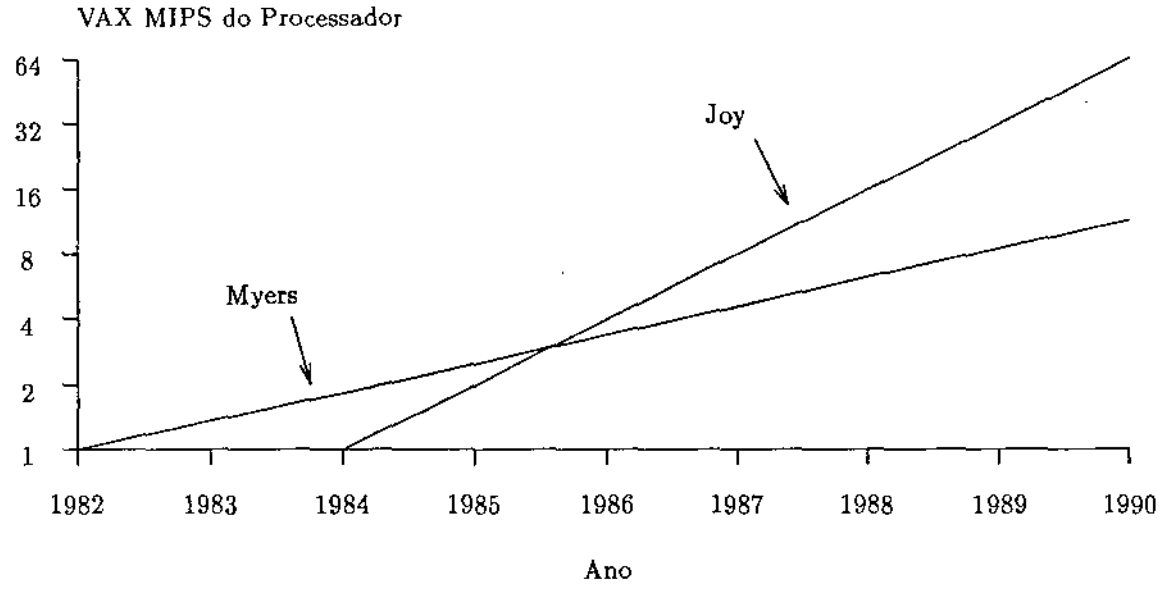

Figura 3.1: Desempenho dos Processadores VLSI

do processador por instruções e dados. Sierwiorek, em [Sie 82], cita que Amdahl relacionou a velocidade da CPU ao tamanho da memória principal usando a seguinte regra:

- cada instrução por segundo da CPU requer um *byte* da memória principal.

Se o custo de um sistema de computação não fosse dominado pelo custo da memória, esta assertiva sugeriria que a capacidade dos *chips* de memória deveria crescer à mesma taxa da velocidade das CPUs. Moore previu essa taxa vinte anos atrás:

$$transistor/chip = 2^{ano-1964}.$$

Como predito pela lei de Moore, as RAMs têm sua densidade quadruplicada a cada dois [Moo 75] ou três anos [Mye 86], como demonstrado na Figura 3.2.

Recentemente, em [Gar 84], a razão entre a capacidade de armazenamento da memória em *MBytes* e a velocidade da CPU em *MIPS* (*millions of instructions per second*) foi denominada *alpha*, com *alpha* = 1 correspondendo à lei de Amdahl. Em parte, devido à rápida queda nos preços das memórias semicondutoras, o tamanho da memória principal tem crescido mais rapidamente que a velocidade da CPU, fazendo com que muitas máquinas sejam vendidas com *alpha* maior ou igual a três.

Capacidade não é a única característica da memória responsável por manter o sistema balanceado. A velocidade com que os dados e instruções são mandados para a CPU também determina seu desempenho. Esta última característica não tem sido um problema por duas razões:

1. *caches*, isto é, uma técnica de organização de memória onde um pequeno *buffer*, na frente de uma memória grande e lenta, pode conter uma fração substancial das referências à memória;

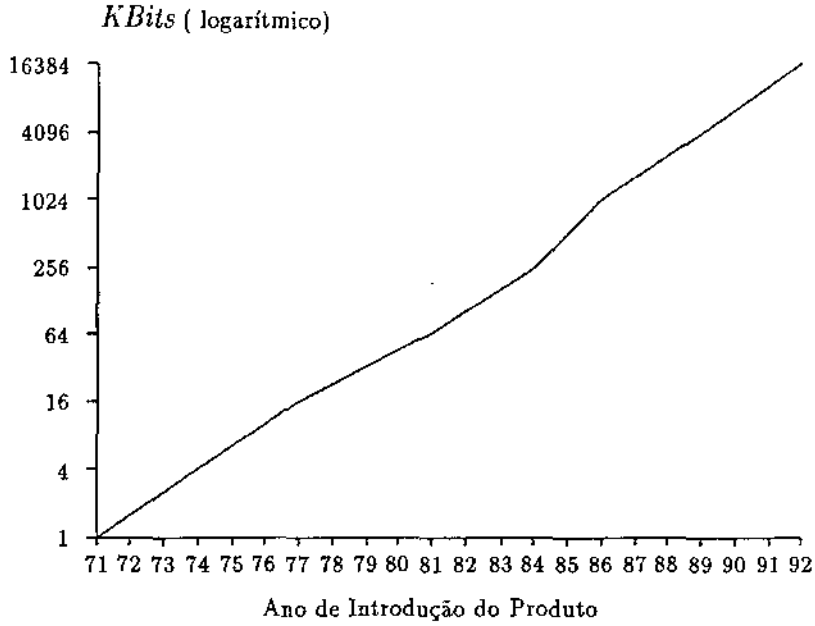

Figura 3.2: Densidade dos *Chips* de Memória por Ano

2. RAMs estáticas, a tecnologia usada para construir os *caches*, cuja velocidade vem aumentando a uma taxa de 40% ou mais por ano.

Para manter balanceado os custos num sistema de computação, o desempenho do armazenamento secundário deve casar com o das outras partes do sistema. Uma medida fundamental na tecnologia dos discos magnéticos é o crescimento do número máximo de *bits* que pode ser armazenado por polegada quadrada. Chamada de MAD (*maximal areal density*), a primeira lei em densidade de discos prediz:

$$MAD = 10^{(ano-1971)/10}.$$

Na Figura 3.3, este valor está plotado para vários discos comerciais. A tecnologia de discos magnéticos tem dobrado de capacidade e reduzido o preço à metade, a cada três anos, de acordo com a taxa de crescimento das memórias semicondutoras.

Em contraste com a tecnologia de memória primária, o desempenho dos discos magnéticos aumentou apenas modestamente. Esses dispositivos mecânicos são dominados pelo *seek time* e *rotation delay*. De 1971 a 1981 o *seek time* de um disco IBM de alto desempenho melhorou apenas de um fator de dois, enquanto que o *rotation delay* não mudou [Har 81]. Na Figura 3.4, vemos a disparidade na evolução das CPUs e dos discos magnéticos. Esta disparidade continua aumentando, e não há razão alguma para se esperar uma melhoria substancial no desempenho dos discos magnéticos num futuro próximo.

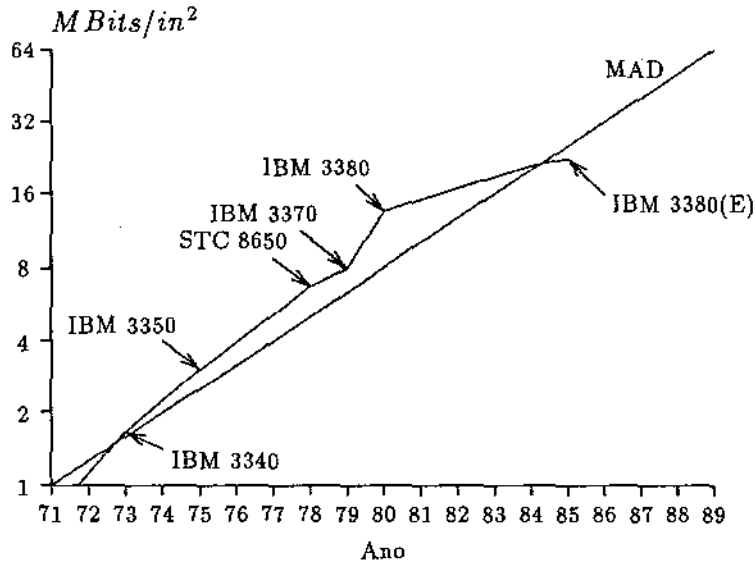

Figura 3.3: Primeira Lei em Densidade de Discos

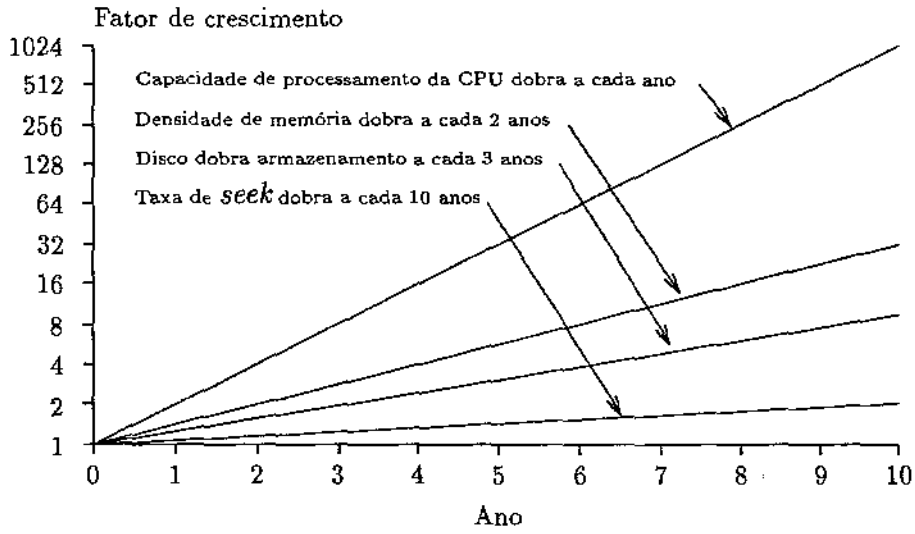

Figura 3.4: Disparidade entre as CPUs e seus Dispositivos de I/O

3.2 Extensões a Arquiteturas Convencionais

Até aproximadamente a metade da década de 80, pouco se fez no sentido de romper com as técnicas tradicionais de construção de subsistemas de disco. Todas as implementações visavam aumentar o desempenho do subsistema ou aumentando o desempenho do dispositivo de armazenamento, ou seja o disco magnético, ou fazendo com que esse dispositivo fosse usado o mínimo possível, via cacheamento ou escalonamento. Existiram, também, tentativas de substituir o armazenamento magnético por armazenamento semicondutor, persistindo, ainda, em alguns sistemas.

As extensões principais, ou seja, aquelas que mais influíram no mercado de informática, são comentadas nas subseções seguintes.

3.2.1 Disco de Cabeças Fixas

Chama-se disco de cabeças fixas (*fixed heads disk*) aquele que possui uma cabeça magnética para cada trilha. Neste tipo de disco, elimina-se o *seek time* mas não há redução nem da latência rotacional nem do tempo de transferência.

Como os discos modernos têm centenas de trilhas por superfície, essa não é uma tecnologia economicamente viável.

3.2.2 Disco de Transferências Paralelas

Este tipo de disco permite que existam diversas leituras e escritas simultâneas em superfícies magnéticas distintas, conseguindo-se, dessa forma, uma taxa de transferência muito alta. Nenhuma melhoria no tempo de posicionamento e na latência rotacional é produzida.

Considerações econômicas e tecnológicas limitam a viabilidade desse tipo de disco. Do ponto de vista econômico, prover mais de um conjunto magnético por atuador é muito caro. Mais ainda, os discos modernos utilizam sofisticados mecanismos de controle para o posicionamento, sendo difícil fazer isto em trilhas de um mesmo cilindro. Parece existir uma contrapartida entre a densidade de trilhas e o número de superfícies: com o aumento da densidade de trilhas, fica mais difícil posicionar em trilhas de superfícies distintas, reduzindo o número de superfícies que podem participar numa transferência paralela. Um subsistema de mais baixo custo pode ser construído usando-se vários discos padrão, em parte devido ao pequeno volume de vendas do disco de transferências paralelas comparado com os discos padrão.

3.2.3 Cacheamento

A idéia por trás desta técnica é manter, em memória principal, os blocos mais recentemente acessados do disco, de modo que acessos repetidos à mesma informação possam ocorrer sem transferências adicionais. Devido à localidade nos acessos aos arquivos, o cacheamento reduz substancialmente as transferências com o disco: nos sistemas atuais, há uma taxa de acerto (*hit rate*) entre 80 e 90% para *caches* de 0.5 a $5 MBytes$. No futuro, haverá *caches* com centenas de $MBytes$, possibilitando manter os arquivos utilizados por dezenas de usuários.

O cacheamento muda a natureza das transferências de I/O com os discos em dois aspectos. Primeiro, ele faz com que as transferências com o disco sejam em sua maior parte para escrita ao invés de para leituras. Segundo, os *caches* provêm um *buffer* para uma seqüência muito rápida de transferências (*burst* I/O).

A parte mais problemática do cacheamento é o gerenciamento das requisições de escrita no disco, que representam aproximadamente 1/3 das operações de *I/O* com arquivos [Ous 85]. Uma possível abordagem seria deixar os dados recentemente escritos no *cache* e postergar a escrita no disco por um determinado intervalo de tempo, a fim de se utilizar o canal de transferência com o disco mais eficientemente. O problema com essa abordagem é que o conteúdo do *cache* pode se perder por algum motivo, como por exemplo falta de energia elétrica.

Uma forma de tornar os *caches* mais seguros é armazená-los em memória protegida por baterias. Além disso, o sistema operacional deve atualizar o sistema de arquivos, com base no conteúdo do *cache*, durante a reinicialização do sistema. Isto, contudo, nem sempre é possível, especialmente se a reinicialização se deveu a uma falha interna ao sistema operacional.

3.2.4 Escalonamento de Transferências

Os atrasos mecânicos, como vistos por um conjunto de requisições de I/O simultâneas, podem ser reduzidos através de um escalonamento efetivo. Por exemplo, o tempo de posicionamento pode ser reduzido se o algoritmo *shortest seek time first* for usado.

A literatura sobre algoritmos de escalonamento é vasta. Observou-se que a eficácia de um algoritmo particular depende criticamente da carga do subsistema, e que esses algoritmos trabalham melhor quando existem longas filas de requisições pendentes. Infelizmente, essa situação é rara em subsistemas reais.

3.2.5 Disco de Estado Sólido

Por razões econômicas[1], discos de estado sólido são usualmente construídos usando-se *chips* de memória relativamente lentos. Assim, podemos encarar esse dispositivo de duas maneiras distintas: ou como sendo um tipo de memória principal lenta, ou como sendo um disco de alta velocidade. Quando vista como memória principal, essa tecnologia é chamada armazenamento expandido (*expanded storage* - ES), a qual é vista pelo *host* como sendo memória e não como sendo dispositivo de I/O[2]. Por exemplo, no IBM 3090, a máxima velocidade de transferência entre o armazenamento expandido e a memória é duas ordens de magnitude mais rápida que os dispositivos convencionais: aproximadamente $216 MBytes/s$.

Se discos de estado sólido forem usados para substituir discos magnéticos, eles devem ser feitos não voláteis. Isto pode ser conseguido via uma bateria, mas a técnica é controvertida. Primeiro, é difícil assegurar que as baterias estejam carregadas quando precisarem ser ativadas. Segundo, é difícil avaliar por quanto tempo deve-se suprir alimentação por bateria para o disco de estado sólido: provavelmente, o tempo de copiar seu conteúdo numa fita magnética.

3.2.6 Sistema de Arquivos Log-Structured

Esta arquitetura de sistema de arquivos [Ous 88] propõe a fusão dos discos de arquivos com o disco que implementa o *cache logging*. A designação *log-structured* vem do fato de que a representação do sistema de arquivos, no disco, é nada mais que um diário (*log*). Quando os arquivos são modificados,

[1] Até o presente momento, existe uma diferença de 10 a 20 vezes no custo por *megabyte* de um disco magnético e de um disco de estado sólido.

[2] O armazenamento expandido é conectado à memória principal via um barramento de alta velocidade, ao invés de por um controlador de I/O.

dados do arquivo juntamente com outras informações são anexadas seqüencialmente ao diário, sem *seeks*, em forma de uma cadeia. Isto permite ao sistema de arquivos utilizar toda a capacidade de transferência do disco, mesmo que os arquivos sejam pequenos[3]. Além desta, tais sistemas de arquivos possuem outras características importantes:

- recuperação rápida: oferecem a possibilidade de recuperação após falhas bem mais rápida que os sistemas de arquivos tradicionais; Sistemas como o UNIX podem deixar as estruturas de controle de alocação do espaço em disco inconsistentes devido a algum tipo de falha; Como parte da operação de reinicialização, o sistema deve vasculhar todas as estruturas de dados pertinentes, a fim de detectar e reparar inconsistências; Este procedimento pode ser muito lento, especialmente quando discos com alguns *GBytes* de capacidade estão presentes no sistema; Em contraste, todas as informações recentemente mudadas num sistema *log-structured* estão na cabeça do diário; É possível organizá-lo de modo que apenas poucos blocos, próximos a sua cabeça, precisem ser examinados durante o processo de recuperação;

- localidade temporal: os arquivos que são escritos aproximadamente no mesmo instante são armazenados aproximadamente na mesma posição do disco; Isto significa que tais grupos de arquivos podem, potencialmente, ser recuperados do disco com um único *seek* seguido de uma leitura longa;

- habilidade em manter versões: a natureza apenas-anexar deste tipo de sistema de arquivos, implica em que versões anteriores dos arquivos modificados estejam presentes no disco; Com um pouco de esforço é possível tornar disponível aos usuários as versões anteriores, a fim de que estes se recuperem de alguns tipos de erros causados por eles mesmos, tais como remoção indevida de algum arquivo.

Num sistema de arquivos tradicional como o UNIX, existem duas fases no processo de localização de um arquivo. Primeiro, seu nome textual deve ser transformado em um identificador interno e, segundo, seu blocos devem ser localizados. Isto é feito, via indexação em uma tabela localizada numa região predeterminada do disco.

Infelizmente, essa nova abordagem não permite a existência de mapas de alocação em regiões fixas, pois isso iria requerer *seeks* a fim de modificar os mapas quando arquivos fossem criados, removidos ou modificados. Ao invés disso, toda informação nova ou modificada, sendo mapeamento ou dados, deve ser escrita na cabeça do diário. Isso pressupõe uma estrutura de mapeamento flutuante[4], integrada com o resto do diário. Existem várias formas de se fazer isto.

A Figura 3.5 ilustra, em três passos, como um sistema de arquivos tradicional, com um único mapa de alocação em posição fixa, pode ser transformado num sistema de arquivos *log-structured* com mapeamento flutuante.

Inicialmente, a Figura 3.5 (a) mostra um disco tradicional com uma matriz de mapeamento em uma área do disco e dados na outra. A Figura 3.5 (b) mostra o primeiro passo em direção a um sistema *log-structured*. Novos blocos de dados são alocados seqüencialmente, a partir da cabeça do diário, com a matriz de mapeamento ainda numa posição fixa.

A Figura 3.5(c) mostra o segundo passo. A matriz de mapeamento é tratada como um arquivo especial, onde seus blocos contêm o mapeamento. Sendo assim, seus blocos podem flutuar como os

[3]Eles podem ser coletados em blocos maiores antes de serem escritos.

[4]Onde a informação de mapeamento não está localizada em uma região predeterminada do disco

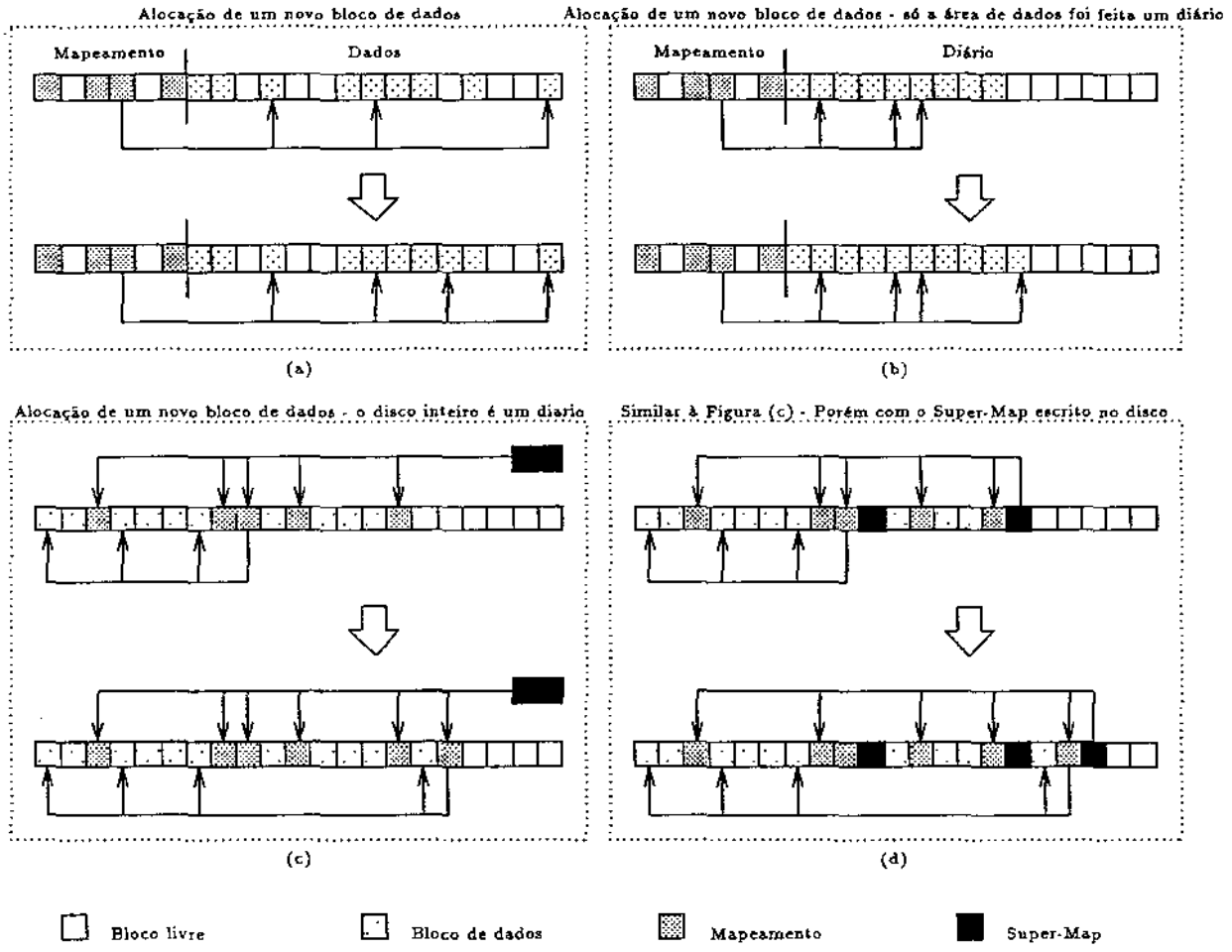

Figura 3.5: Derivação de um Sistema de Arquivos *Log-Structured* a partir de um Sistema de Arquivos Tradicional

de qualquer outro arquivo: quando uma entrada na matriz é alterada, o respectivo bloco do arquivo é rescrito na cabeça do diário. Um mapeamento adicional chamado *super-map* dá a localização de cada bloco do arquivo de mapeamento. Cada referência a uma entrada na matriz de mapeamento deve consultar o *super-map* para localizar a entrada no disco.

O passo final consiste em, de vez em quando, escrever o *super-map* na cabeça do diário. Isto está ilustrado na Figura 3.5 (d). O *super-map* deve conter informações que o identifique, de forma não ambigüa. Após a queda do sistema, tudo que é necessário para recuperar o sistema de arquivos é procurar para trás, no diário, a cópia mais recente do *super-map*.

3.3 Novas Propostas de Arquitetura

Após a metade da década de 80, muitos trabalhos inovadores foram publicados no âmbito dos subsistemas de discos magnéticos. Ao contrário das extensões a arquiteturas convencionais, não se pretende melhorar o desempenho desses subsistemas melhorando-se o desempenho dos discos. O que se tenta é construir um disco lógico composto de diversos discos físicos de modo a tirar proveito do possível paralelismo de leituras e escritas. Os discos físicos podem estar na mesma máquina ou em máquinas distintas conectadas por uma rede local. Todas as novas propostas de arquitetura, além da melhoria de desempenho propiciada, provêm tolerância à falhas.

As novas propostas principais são comentadas nas subseções seguintes.

3.3.1 Matrizes de Disco sem Redundância

A primeira idéia que rompeu com as abordagens tradicionais foi agrupar um conjunto de discos e apresentá-los às aplicações como sendo um disco lógico único. Desta forma, os *bandwidths* dos diversos discos poderiam ser usados para servir uma única requisição lógica de I/O, como também, existiria a possibilidade de haver múltiplas operações de I/O em paralelo. Três arquiteturas, com essa idéia básica, foram idealizadas. Todas serão abordadas nessa seção.

As arquiteturas que serão vistas constituíram os embriões das arquiteturas RAID, descritas na próxima seção. Nada comercial foi produzido usando essas idéias. Praticamente falando, o único uso aceitável das matrizes sem redundância é em aplicações que, por sua natureza, suportam erros em seus dados de entrada, tais como as de processamento de imagens.

As três arquiteturas propostas podem ser classificadas, segundo os seguintes aspectos independentes:

1. independência de atuadores: os atuadores dos elementos da matriz podem ser posicionados como uma unidade ou individualmente;

2. independência de rotação: os rotores dos discos giram juntos, de modo que o mesmo setor está passando pelas cabeças de todos os discos, ou giram independentemente.

Isso porque, para as matrizes sem redundância, só foi proposto entrelaçamento a nível de setor.

Entrelaçamento Síncrono com Movimento dos Braços como Unidade

A idéia básica por trás dessa organização é fazer a matriz se apresentar ao *host* como um disco único de grande capacidade de armazenamento e alta taxa de transferência de dados. Aqui, um setor lógico é entrelaçado entre os diversos discos, de modo similar ao entrelaçamento de um subsistema de

memória. Por exemplo, considere uma transferência de um bloco com M *bytes* que está espalhado entre N discos. O bloco lógico será decomposto em N blocos físicos de M/N *bytes* que serão colocados nos discos físicos.

Nessa arquitetura, o escalonamento fica simplificado, pois apenas uma fila de I/O precisa ser mantida. Assim, o tempo de transferência para um I/O lógico fica reduzido por um fator de N. O *seek time* e o *rotation delay* não se alteram.

Um problema potencial é que basta um disco falhar para que todos os dados da matriz se tornem inacessíveis. Outro problema é como tratar os setores ruins (*bad sectors*). A abordagem tradicional é remapear as áreas ruins para outra porção do disco, reservada para este propósito. Contudo, uma área remarcada gera ineficiência, pois cada braço tem que se mover para acessar os dados da maioria dos discos e se reposicionar para acessar os dados remapeados. Felizmente, a maioria das áreas ruins são detectadas durante a manufatura dos discos. A melhor solução é marcar como ruim a união das áreas ruins, detectadas em cada disco individual. Teremos também problemas se o *clock* de sincronização falhar.

Entrelaçamento Assíncrono com Movimento dos Braços como Unidade

Essa organização é similar à anterior, mas não requer que a rotação dos discos seja sincronizada. Ela mantém todas as vantagens anteriores, elimina o problema do *clock* de sicronização e elimina o problema dos setores ruins. Tais matrizes são mais fáceis de serem construídas.

Como as rotações não são sincronizadas, o desempenho da matriz tende para a do pior disco [Kim 87]. Em outras palavras, a latência rotacional da matriz tende para o pior caso, ao invés de tender para o caso médio.

Entrelaçamento Assíncrono com Movimento Independente dos Braços

Esta organização, conhecida como *declustering* [Liv 87], ao contrário das anteriores, suporta várias operações independentes de I/O. Assim, um acesso seqüencial bem grande pode ser quebrado em múltiplos acessos paralelos que são escalonados independentemente. O problema é a complexidade adicional de se ter múltiplas filas de requisição, e a não confiabilidade de se espalhar um arquivo entre múltiplos discos.

3.3.2 Matrizes Redundantes de Discos Baratos (RAID)

O primeiro trabalho sobre o tema foi publicado em 1988 por Patterson, Gibson e Katz, todos pesquisadores da Universidade da Califórnia em Berkeley, sob o título *A Case for Redundant Array of Inexpensive Disks (RAID)* [Pat 88]. Este artigo estabeleceu cinco tipos de arquitetura, chamados níveis de RAID, e abordou os prós e contras de cada um, quando comparados ao SLED (*single large expensive disk*) e quando comparados entre si. Citou também que todas as idéias ali contidas poderiam ser implementadas tanto em *hardware* como em *software*. Como, efetivamente, uma matriz de discos aumenta o desempenho, depende do nível de RAID empregado e da implementação do fabricante. Matrizes de disco geralmente aumentam o desempenho suportando múltiplas operações simultâneas de leitura e escrita por grupo de discos.

A controladora que é usada para comandar os discos da matriz afeta, de maneira crucial, o desempenho e a tolerância à falhas do subsistema. Como as matrizes de discos são normalmente implementadas fazendo uso de uma única controladora, se esta viesse a falhar, o subsistema de disco

estaria inoperante. Para contornar essa situação, duas ou mais controladoras podem ser usadas numa implementação, permitindo a operação do subsistema no caso de haver falha em uma delas. Existe outra vantagem: múltiplas controladoras provêm canais de dados adicionais aos discos e, conseqüentemente, aumentam a taxa de transferência da matriz.

Controladoras SCSI (*small computer system interface*) são usadas em quase todas as implementações comerciais devido à inteligência associada a essa interface. Uma característica, particularmente importante, da SCSI é sua capacidade de desconexão/reconexão, onde uma controladora pode enviar um comando para um disco em particular, e então desconectar-se dele, antes que o comando seja completado, a fim de enviar um comando para outro disco, para execução paralela. Posteriormente, a controladora reconecta-se com cada disco a fim de completar o trabalho iniciado em cada um deles.

Redundância é uma característica fundamental nas modernas matrizes de disco. O *software driver* da matriz ou a controladora deve ser capaz de detectar uma falha num disco, notificar o operador do sistema e ainda continuar a processar as requisições de leitura e escrita. Em todos os níveis de RAID, utiliza-se fortemente a hipótese de que é altamente improvável, quase impossível, que haja mais que uma falha em um grupo durante o intervalo de tempo de reparação da matriz.

Para eliminar o tempo de queda do sistema, devido à falhas na matriz, as implementações permitem o *hot swapping* dos discos com falha, ou seja, a reposição de um disco com falha por um sem falha com o subsistema servindo normalmente as requisições. Após a troca, os dados perdidos são reconstruídos no novo disco. Os usuários só sentem uma pequena queda de desempenho do subsistema[5].

Confiabilidade

A abordagem básica é quebrar a matriz em grupos[6] de confiabilidade, cada qual possuindo *check disks* extras que contêm informação redundante. Quando um disco falha, supomos que dentro de um pequeno intervalo de tempo, este será reposto e a informação original será reconstruída no novo disco usando-se a informação de redundância. Este tempo é chamado *mean time to repair* (MTTR). O MTTR pode ser reduzido se o sistema incluir discos extras para agirem como *hot standby* — quando um disco falha, um disco de reposição é selecionado de modo automático. Periodicamente, um operador humano repõe todos os discos com defeito.

Usa-se a seguinte convenção:

- D = número total de discos com dados (não incluindo os *check disks*);
- G = número de discos com dados em um grupo (não incluindo os *check disks*);
- C = número de *check disks* em um grupo;
- n_G = D/G = número de grupos.

Em [Pat 88] mostra-se que o MTTF da organização RAID é dada por

$$MTTF_{RAID} = \frac{(MTTF_{Disco})^2}{(D + C * n_G) * (G + C - 1) * MTTR},$$

supondo-se que se consertam todas as falhas simples e que a taxa de falhas seja constante.

[5] Devido à tarefa de reconstrução que está operando em *background*

[6] Um grupo é a mínima coleção de discos sobre a qual a informação de redundância é computada.

Overhead Devido à Confiabilidade

O *overhead* devido à confiabilidade é simplesmente a quantidade de *check disks* extras, expressados como uma percentagem do número total de discos.

Percentagem de Dados Utilizáveis

A percentagem de dados utilizáveis é simplesmente a quantidade de discos de dados, expressados como uma percentagem do número total de discos.

Desempenho

Como aplicações de supercomputadores e de sistemas de processamento de transações têm diferentes padrões de acesso e diferentes taxas de transferência, precisamos de métricas diferentes para se obter avaliações confiáveis. Para supercomputadores, podemos contar o número de leituras e escritas por segundo de grandes blocos de dados, onde grande significa pelo menos um setor de cada disco de dados num grupo. Durante tais transferências, todos os discos num grupo agem como se fossem um único dispositivo.

Uma métrica mais adequada para o processamento de transações é o número de leituras ou escritas individuais por segundo.

Em suma, aplicações de supercomputadores requerem uma alta taxa de dados, enquanto processamento de transações uma alta taxa de I/O.

RAID Nível 1: Discos Espelhados

O uso de espelhamento (*mirroring*) na tolerância à falhas em redes de microcomputadores foi bem estabelecido antes de receber a designação RAID 1. Ainda está em uso comercial, sendo a Novell Netware líder de mercado nesse segmento.

A técnica de discos espelhados(*mirrored disks*) é uma abordagem tradicional para se aumentar a confiabilidade dos discos magnéticos. Esta é a opção mais cara de todas, pois todos os discos são duplicados($G = 1$ e $C = 1$). A Tabela 3.1 mostra as métricas para esse nível de RAID [Pat 88]. Na Tabela 3.1, e nas subseqüentes, RMW significa *read-modify-write*.

Quando acessos individuais são distribuídos entre múltiplos discos, os tempos médios de enfilamento, *seek* e rotação podem diferir de um único disco. Embora o *bandwidth* possa ficar inalterado, fica mais distribuído, reduzindo a variância no tempo de enfilamento e, se a carga não for muito alta, reduzindo a expectativa de tempo de enfilamento via paralelismo [Liv 87]. Quando muitos braços têm que se posicionar sobre a mesma trilha e, então, esperar pela passagem do setor desejado, o tempo médio de *seek* e o tempo médio de rotação serão maiores que os tempos correspondentes para um único disco, tendendo para os tempos de pior caso. Este efeito, em geral, não chega a dobrar os tempos médios de acesso. No caso especial de discos espelhados com controladoras suficientes, a escolha dos braços que podem participar do acesso ao disco pode reduzir os tempos médios em aproximadamente 45% [Bit 88].

Devido às considerações citadas, utiliza-se um fator de redução S quando existem mais de dois discos no grupo. Em geral, $1 \leq S \leq 2$ sempre que grupos de discos trabalhem em paralelo. Para discos síncronos, $S = 1$.

No espelhamento, dois discos armazenam informações idênticas, de modo que um é a cópia (espelho) do outro. Como conseqüência, a cada operação de escrita, o subsistema deve escrever nos

MTTF	4500000 Hs	
Número de Discos	2D	
Custo do *Overhead*	100%	
Capacidade Útil	50%	
Eventos	Eventos/Seg	Eficiência por Disco
Leituras Grandes (ou Agrupadas)	2D/S	1.00/S
Escritas Grandes (ou Agrupadas)	D/S	.50/S
RMWs Grandes (ou Agrupadas)	4D/3S	.67/S
Leituras Pequenas (ou Individuais)	2D	1.00
Escritas Pequenas (ou Individuais)	D	.50
RMWs Pequenas (ou Individuais)	4D/3	.67

Tabela 3.1: Características do RAID nível 1

dois discos, e a cada operação de leitura, o subsistema pode ler de qualquer um deles. Ocorrendo uma falha em um dos discos, o outro continua operando, até que o primeiro seja reposto, sem perda de dados e sem *downtime*[7].

Como a operação de escrita degrada o desempenho do subsistema, a maioria das implementações comerciais emprega *duplexing*, onde cada disco de um grupo possui sua própria controladora. Essa abordagem permite que a atualização dos elementos do grupo se dê em paralelo, aumentando o desempenho do subsistema de discos. Também, temos o aumento da confiabilidade, pois a operação se manteria ininterrupta na falha de uma das controladoras.

Mesmo provendo boa tolerância à falhas, essa arquitetura é cara de ser implementada, pois apenas metade da capacidade de armazenamento da matriz é útil a nível de usuário.

RAID Nível 2: Código Hamming para Correção de Erros

A experiência com organizações de memória principal sugeriu uma forma de reduzir o custo da confiabilidade. Com a introdução das memórias DRAM de 4K x 1 e 16K x 1, foi descoberto que tais dispositivos são sujeitos a interferência devido às partículas alfa. A solução encontrada para o problema da confiabilidade foi adicionar ao grupo de dados (normalmente com largura de 8, 16, 32 ou 64 *bits*) *chips* de redundância para corrigir erros simples e detectar erros duplos em cada grupo.

Como todos os bits de dados são lidos e escritos simultaneamente, não há impacto no desempenho. Contudo, leituras menores que o tamanho do grupo requerem a leitura de todo ele para se assegurar que a informação esteja correta, e escritas de uma porção do grupo requerem três passos:

1. leitura para se obter o resto dos dados;
2. modificação para adicionar a nova informação à anterior;
3. escrita de todo o grupo, incluindo a informação de redundância.

Na organização RAID nível 2, tenta-se imitar a solução para as DRAMs, intercalando-se a nível de bit os dados entre os discos de um grupo, e então adicionando-se um número suficiente de discos de redundância para detectar e corrigir falhas simples, e para detectar falhas duplas.

[7]Tempo de parada do sistema devido a uma falha não consertada

Devido à quantidade de informação de redundância requerida, vários *check disks* podem ser necessários para se implementar essa arquitetura. Por exemplo, em uma matriz com 10 discos de dados, precisaríamos de quatro *check disks*, resultando em aproximadamente 71% de uso útil da matriz. Uma matriz com menos de 10 discos de dados se torna impraticável, devido à queda do espaço de armazenamento útil, que pode atingir menos de 50%.

Como a unidade de transferência num disco magnético é um setor, temos que uma transferência em RAID 2 envolve no mínimo G setores.

A Tabela 3.2 mostra as métricas para essa organização [Pat 88]. Nesta, e nas demais tabelas sobre RAID, Ni (i=1,...,5) significa RAID nível i e Ni/Nj significa a comparação entre o nível i relativamente ao nível j.

		G=10		G=25	
MTTF		> Vida Útil		> Vida Útil	
Número Total de Discos		1.4D		1.2D	
Custo do *Overhead*		40%		20%	
Capacidade Útil		71%		83%	
Eventos	Eventos/Seg	Eficiência por Disco		Eficiência por Disco	
		N2	N2/N1	N2	N2/N1
Leituras Grandes	D/S	0.71/S	71%	0.86/S	86%
Escritas Grandes	D/S	0.71/S	143%	0.86/S	172%
RMWs Grandes	D/S	0.71/S	107%	0.86/S	129%
Leituras Pequenas	D/SG	0.07/S	6%	0.03/S	3%
Escritas Pequenas	D/2SG	0.04/S	6%	0.02/S	3%
RMWs Pequenas	D/SG	0.07/S	9%	0.03/S	4%

Tabela 3.2: Características do RAID nível 2

Devido ao código Hamming entrelaçado, RAID 2 é relativamente complexo de se implementar. É ótimo para leitura e escrita de grandes blocos de dados a uma alta taxa de transferência. Para este tipo de requisição, RAID 2 apresenta o mesmo desempenho que RAID 1, mesmo utilizando menos discos. Contudo, leituras e escritas de pequenos blocos são ineficientes, e a operação de *read-modify-write*, requerida para escrita de pequenos blocos, resulta em baixa performance. Assim RAID nível 2 é adequado para supercomputadores, mas inapropriado para sistemas de processamento de transações.

Enquanto implementações para *mainframe* foram desenvolvidas, RAID 2 é impraticável para pequenos sistemas.

RAID Nível 3: Check Disk Único por Grupo

A arquitetura RAID 3 é utilizada largamente em *workstations* e está experimentando um rápido avanço em redes de microcomputadores.

RAID 3 utiliza um único *check disk*, algumas vezes chamado de disco de paridade (*parity disk*), para cada grupo de discos. Os dados escritos são entrelaçados a nível de *bit* entre os discos de dados. O *check disk* recebe o XOR (ou exclusivo) de todos os dados escritos nos discos de dados.

Como transferências de e para um disco individual ocorre em unidades múltiplas de setores, a mínima quantidade de dados que pode ser lida ou escrita da matriz é o número de discos de dados multiplicado pelo número de *bytes* em um setor. Isto é conhecido como *unidade de transferência*.

RAID 3 utiliza apenas um *check disk* pois os discos de microcomputadores já incorporam códigos corretores de erros em cada setor, de modo que a técnica de recuperação de erros usada por RAID 2 é desnecessariamente redundante se a matriz for construída com tais discos. Apenas informação suficiente para recuperação dos dados do disco com falha precisa ser armazenada na matriz. A informação do disco com falha pode ser reconstruída calculando-se a paridade dos discos bons restantes e então comparando-se bit a bit com a paridade original do grupo. Quando as duas paridades coincidem, o bit com falha era 0, caso contrário era 1. Se a falha ocorreu no *check disk*, calcula-se novamente a paridade e armazena-se esse valor no disco de reposição.

A Tabela 3.3 sumariza as características do RAID nível 3 [Pat 88].

		G=10			G=25		
MTTF		> Vida Útil			> Vida Útil		
Número Total de Discos		1.1D			1.04D		
Custo do *Overhead*		10%			4%		
Capacidade Útil		91%			96%		
Eventos	Eventos/Seg	Eficiência por Disco			Eficiência por Disco		
		N3	N3/N2	N3/N1	N3	N3/N2	N3/N1
Leituras Grandes	D/S	0.91/S	127%	91%	0.96/S	112%	96%
Escritas Grandes	D/S	0.91/S	127%	182%	0.96/S	112%	192%
RMW Grandes	D/S	0.91/S	127%	136%	0.96/S	112%	142%
Leituras Pequenas	D/SG	0.09/S	127%	8%	0.04/S	112%	3%
Escritas Pequenas	D/2SG	0.05/S	127%	8%	0.02/S	112%	3%
RMW Pequenas	D/SG	0.09/S	127%	11%	0.04/S	112%	5%

Tabela 3.3: Características do RAID nível 3

Como múltiplos discos da matriz podem ser lidos ou escritos simultaneamente, taxas de transferência de dados extremamente altas podem ser conseguidas. Isto é particularmente verdadeiro quando o tamanho dos dados que estão sendo lidos ou escritos é maior ou igual ao tamanho da unidade de transferência. O desempenho do RAID 3 diminui consideravelmente quando ocorrem muitas transferências pequenas, como em uma aplicação de processamento de transações. Para leituras menores que a unidade de transferência, todos os dados em uma unidade de transferência devem ser lidos de qualquer forma. Isto torna a operação de leitura maior do que seria necessário, reduzindo a eficiência da operação.

A ineficiência na escrita é ainda pior. Para escritas menores que a unidade de transferência, apenas uma porção de um setor de cada disco de dados deve ser modificada. Mas, novamente, o disco deve lidar com transferências de unidades inteiras. Neste caso, uma unidade de transferência deve ser lida do disco; os dados de escrita devem ser sobrepostos aos dados lidos, onde for apropriado; os dados são escritos de volta aos discos, com o *check disk* sendo atualizado apropriadamente.

Este esquema certamente resulta em operações de escrita lentas para pequenas quantidades de dados. Por causa disso, implementações do RAID 3 são preferidas para aplicações que processem

grandes quantidades de dados, tais como ocorrem em supercomputadores.

A latência rotacional (*rotational latency*) pode variar entre os discos da matriz, resultando em acúmulo de atrasos durante a operação de transferência. Para eliminar esse problema, a maioria das implementações do RAID 3 incorporam sincronização de rotação, para máximo desempenho.

RAID Nível 4: Leituras e Escritas Independentes

Espalhar as transferências entre todos os discos no grupo é vantajoso pois:

- o tempo de transferência é reduzido devido ao *bandwidth* da matriz;
- operação de leitura e escrita num disco de um grupo requer leituras/escritas de todos os discos no grupo; Níveis 2 e 3 de RAID podem fazer apenas um I/O em dado instante por grupo;
- se os discos não estiverem sincronizados, não se vê um tempo médio de *seek time* e *rotation delay*; Os tempos observados movem-se em direção ao do pior caso.

Este nível de RAID aumenta o desempenho das pequenas transferências através do paralelismo. A informação agora não está mais espalhada entre diversos discos; cada unidade individual está em um único disco.

Ao se guardar a unidade de transferência em um setor, as leituras podem ser independentes e operar à máxima taxa de leitura do disco e ainda detectar erros. Assim, a principal diferença entre os níveis 3 e 4 é que o *interleave* de dados é feito a nível de setor, ao invés de bit.

Como em RAID 3, RAID 4 dedica um disco para guardar a informação de redundância. Enquanto essa abordagem permite múltiplas leituras em paralelo, as escritas são o gargalo do subsistema. Como um único *check disk* deve ser escrito a cada operação de escrita, apenas uma operação de escrita pode estar ocorrendo em um dado instante.

Em RAID nível 4, o que não ocorre no nível 3, o cálculo da paridade é muito simples pois, se conhecermos o valor anterior do dado e da paridade, bem como o novo valor, podemos calcular a nova paridade através da expressão:

$$\textit{nova paridade} = (\textit{dado anterior}\ \textbf{xor}\ \textit{dado novo})\ \textbf{xor}\ \textit{paridade anterior}$$

A Tabela 3.4 sumariza as características do RAID nível 4 [Pat 88]. Nota-se que todos os pequenos acessos melhoraram (dramaticamente para leitura) mas os pequenos acessos *read-modify-write* continuam muito lentos em relação ao nível 1, de modo que sua aplicação em sistemas de processamento de transações é duvidoso.

RAID Nível 5: Paridade Rotacionada

Enquanto o RAID nível 4 possui paralelismo para leituras, as escritas são ainda limitadas a uma por grupo, pois toda escrita deve ler e escrever no *check disk*.

RAID 5 elimina o problema com a escrita no subsistema RAID 4. Ao contrário do RAID 4 que dedica um disco inteiro para a informação de redundância, RAID 5 dedica o equivalente de um disco inteiro, mas distribui os dados de redundância entre todos os discos do grupo. O impacto no desempenho devido a essa pequena mudança é muito grande pois o RAID nível 5 pode suportar múltiplas escritas individuais por grupo.

Deixamos para o Capítulo 4 a apresentação detalhada desta arquitetura. Contudo, sumarizamos na Tabela 3.5 as características desse nível de RAID [Pat 88].

		G=10			G=25		
MTTF		> Vida Útil			> Vida Útil		
Número Total de Discos		1.1D			1.04D		
Custo do *Overhead*		10%			4%		
Capacidade Útil		91%			96%		
Eventos	Eventos/Seg	Eficiência por Disco			Eficiência por Disco		
		N4	N4/N3	N4/N1	N4	N4/N3	N4/N1
Leituras Grandes	D/S	0.91/S	100%	91%	0.96/S	100%	96%
Escritas Grandes	D/S	0.91/S	100%	182%	0.96/S	100%	192%
RMW Grandes	D/S	0.91/S	100%	136%	0.96/S	100%	146%
Leituras Pequenas	D	0.91	1200%	91%	0.96	3000%	96%
Escritas Pequenas	D/2SG	0.05	120%	9%	0.02	120%	4%
RMW Pequenas	D/G	0.09	120%	14%	0.04	120%	6%

Tabela 3.4: Características do RAID nível 4

		G=10			G=25		
MTTF		> Vida Útil			> Vida Útil		
Número Total de Discos		1.1D			1.04D		
Custo do *Overhead*		10%			4%		
Capacidade Útil		91%			96%		
Eventos	Eventos/Seg	Eficiência por Disco			Eficiência por Disco		
		N5	N5/N3	N5/N1	N5	N5/N3	N5/N1
Leituras Grandes	D/S	0.91/S	100%	91%	0.96/S	100%	96%
Escritas Grandes	D/S	0.91/S	100%	182%	0.96/S	100%	192%
RMW Grandes	D/S	0.91/S	100%	136%	0.96/S	100%	144%
Leituras Pequenas	(1+C/G)D	1.0	110%	100%	1.0	104%	100%
Escritas Pequenas	(1+C/G)D/4	0.25	550%	50%	0.25	1300%	50%
RMW Pequenas	(1+C/G)D/2	0.5	550%	75%	0.5	1300%	75%

Tabela 3.5: Características do RAID nível 5

3.3.3 Espelhamento Distorcido

A técnica de espelhamento distorcido (*distorted mirrors*) foi proposta por J. A. Solworth e C. U. Orji, ambos pesquisadores da Universidade de Illinoi em Chicago, num artigo publicado em 1991 sob o título *Distorted Mirrors* [Sol 91].

Nesta abordagem, como no espelhamento tradicional (RAID 1), há, de fato, replicação de um disco lógico entre os discos físicos do conjunto de espelhamento. Mas, ao contrário de RAID 1, as imagens em cada membro do conjunto não são iguais: cada qual contém os mesmos blocos lógicos, mas não na mesma ordem.

Suponha que existam B blocos no disco lógico. Nesta técnica, como pode ser visto na Figura 3.6, o i-ésimo disco físico do conjunto de espelhamento ($0 \leq i \leq 1$) possui todos os B blocos particionados em duas zonas distintas:

1. a *zona de mapeamento fixo* que contém os blocos lógicos $i\frac{B}{2}..(i+1)\frac{B}{2}-1$ em seqüência, e
2. a *zona de mapeamento distorcido* que contém os demais blocos; Estes últimos não possuem uma ordem particular de escrita no disco.

Solworth sugeriu particionamento a nível de cilindro. Isto significa que cada cilindro contém blocos da zona fixa, conhecida como partição mestre (*master partition*), e da zona distorcida, conhecida como partição escrava (*slave partition*).

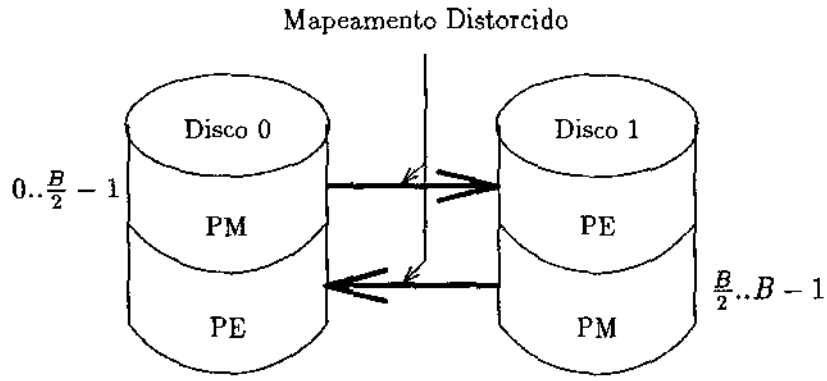

Figura 3.6: Espelhamento Distorcido

Cada escrita lógica ocorre numa posição fixa da partição mestre e em qualquer posição livre da partição escrava. Já, uma requisição de leitura pode ser servida via a partição fixa ou via a escrava. Contudo, leituras seqüenciais multibloco são mais eficientes se servidas via a partição mestre, pois esses blocos, em geral, não estão seqüencialmente alocados na partição escrava.

A principal vantagem do espelhamento distorcido é que a maioria das escritas escravas podem ser feitas sem *seek time* e com *rotational delay* reduzido.

Na implementação do espelhamento distorcido, duas estruturas de dados desempenham um papel fundamental no controle da política de alocação:

1. o `FreeMap`, um bit map com 1 para cada bloco escravo alocado, e

2. o `DistortedMap`, uma função cujo domínio são as posições lógicas, e a imagem as posições das cópias escravas dos blocos lógicos.

O `FreeMap` é necessário para encontrar posições onde se podem escrever os blocos escravos, e o `DistortedMap` é utilizado apenas para atualizar o `FreeMap`.

Quando o bloco lógico l é alterado, ocorre a seguinte seqüência de eventos:

1. `DistortedMap[`l`]` é adicionado ao `FreeMap`;

2. um bloco é escolhido do `FreeMap` para armazenar o bloco escravo;

3. `DistortedMap[`l`]` recebe o endereço do bloco selecionado;

4. `DistortedMap[`l`]` é removido do `FreeMap`.

Para assegurar que essa operação sempre seja possível, deve-se garantir que exista um número suficiente de posições livres na partição escrava. Isto só é possível se supuzermos um fator de utilização inferior a 100% (tipicamente 90%). Supondo que x seja o fator de utilização de um espelhamento distorcido com B blocos, cada disco é particionado como se segue: $xB/2$ blocos por disco são dedicados à partição mestre, enquanto que o restante vai para a partição escrava.

Até aqui supomos, implicitamente, a operação sob condições normais. Isto é, sem erros. Contudo, esta arquitetura está preparada para contornar dois tipos de falhas: falhas de disco e falhas de controladora.

No caso de falha de controladora, os dados em cada disco não foram corrompidos, mas o `DistortedMap` e o `FreeMap` foram perdidos e devem ser reconstruídos. A solução para esse problema é baseado na técnica de *backup* incremental dessas duas estruturas.

Havendo falha de disco, todos os blocos continuam acessíveis pois o `DistortedMap` e o `FreeMap` continuam válidos. Normalmente, existe um terceiro disco no qual é reconstruído o disco com falha, a fim de manter a confiabilidade do sistema. Este procedimento é mais lento que em RAID 1, pois devido ao mapeamento distorcido, não se pode copiar o conteúdo de um disco no outro diretamente.

3.3.4 A Arquitetura Swift

A arquitetura Swift foi proposta por L. F. Cabrera, do centro de pesquisas da IBM em Almaden, e por D. D. E. Long, da Universidade da Califórnia em Santa Cruz, em 1990, em um relatório técnico sob o título *Swift: A Storage Architecture for Large Objects* [Cab 90]. Swift tenta aumentar a taxa de transferência de grandes objetos, armazenados em dispositivos lentos. As aplicações vão desde visualização em sistemas de CAD (*computer aided design*) até processamento de imagens coloridas em tempo real.

A arquitetura Swift distingue quatro componentes lógicos: o agente de distribuição (*distribution agent*), o produtor de dados (*data producer*), o mediador de armazenamento (*storage mediator*) e o agente de armazenamento (*storage agent*). Todos conectados via uma rede de alta velocidade.

O agente de distribuição opera em estreita cooperação com o produtor de dados e é responsável pelo empacotamento e transmissão dos dados aos agentes de armazenamento. Ele obtém do mediador de armazenamento serviços de diretório, permissões de acesso, chaves de encriptação, e planos de armazenamento e recuperação da informação. O agente de distribuição é também responsável por implementar o protocolo especificado no plano de armazenamento.

O mediador de armazenamento coopera com os agentes de armazenamento para prover serviços de diretório e autenticação, para produzir planos de armazenamento, para gerenciar a coerência do *cache* e para assegurar a serializabilidade das atividades concorrentes. Os agentes de armazenamento comandam os dispositivos de armazenamento. São responsáveis pelo armazenamento e recuperação dos dados a uma taxa de transferência negociada, e por manter informação de redundância suficiente para permitir reconstrução dos objetos se o mediador de armazenamento falhar. Uma possível implementação da arquitetura Swift é mostrada na Figura 3.7.

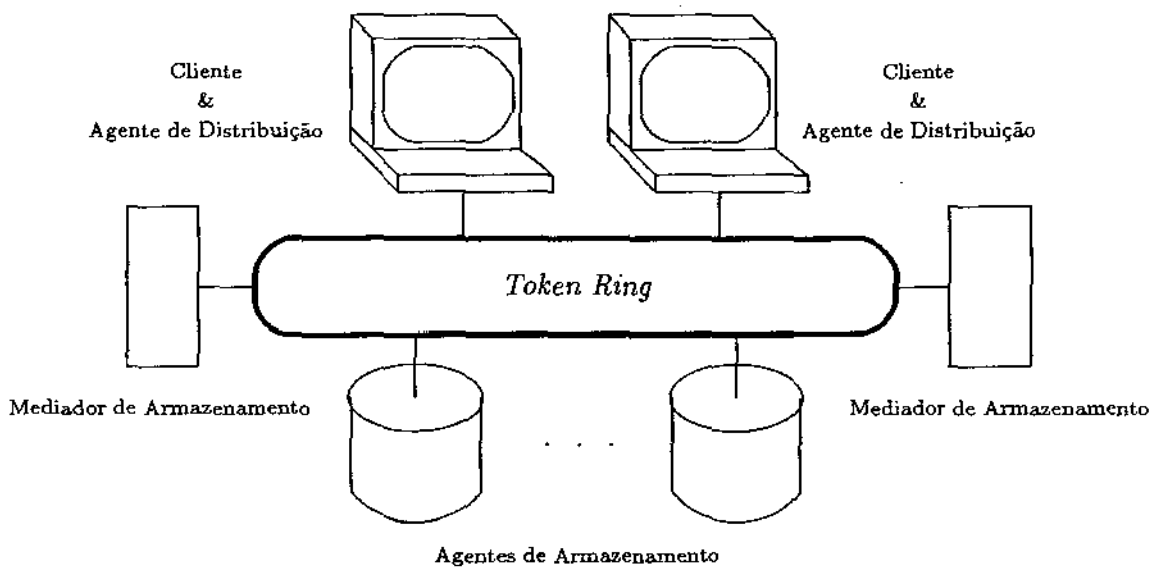

Figura 3.7: Uma Configuração para a Arquitetura Swift

Para armazenar um objeto usando-se Swift, o produtor de dados contacta o agente de distribuição informando o nome, o tamanho estimado e a taxa de transferência para o objeto a ser armazenado. Este, então, contacta o mediador de armazenamento com os mesmos parâmetros.

O mediador de armazenamento determina o grau de redundância requerido para o objeto e qual conjunto de agentes de armazenamento servirá a requisição. Quando a inicialização da requisição estiver completa, cada agente de armazenamento terá reservado os recursos necessários em termos de armazenamento e capacidade de transferência para sua parte do objeto. O mediador de armazenamento retorna uma coleção de *handles* para as partes do objeto ao agente de distribuição, como parte do plano de armazenamento para o objeto. Até o fim da transmissão, o agente de distribuição transfere os dados diretamente aos agentes de armazenamento, de forma entrelaçada, e não faz uso do mediador de armazenamento como intermediário.

O cenário para a recuperação de dados é análogo, com a única diferença sendo que o plano diz

ao agente de distribuição de quais agentes de armazenamento os dados devem ser recuperados. A colagem e a apresentação ao cliente é feita pelo agente de distribuição.

Um problema com essas abordagens é falha parcial. Soluciona-se através de múltiplas cópias de objetos ou através de códigos corretores de erros. A arquitetura Swift é flexível e pode aplicar ambos os métodos.

3.3.5 O Sistema de Arquivos Distribuído Zebra

A arquitetura Zebra foi proposta por J. H. Hartman e J. K. Ousterhout, ambos da Universidade da Califórnia em Berkeley, num artigo publicado em 1992 sob o título *Zebra: A Striped Network File System* [Har 92].

Zebra é um sistema de arquivos distribuído projetado para prover alto desempenho e alta disponibilidade dos dados, que incorpora técnicas de sistemas *log-structured* e tecnologia RAID. Dos sistemas *log-structured*, Zebra utiliza a idéia de que pequenas escritas independentes podem ser agrupadas em grandes escritas seqüenciais, aumentando o desempenho do subsistema de armazenagem. Como em RAID, Zebra utiliza entrelaçamento e paridade, resultando num sistema de arquivos distribuído que entrelaça os dados entre múltiplos servidores de armazenamento, usa paridade para prover grande disponibilidade, e transfere dados dos arquivos entre os clientes e os servidores de armazenamento em grandes unidades. As características mais marcantes de Zebra são as seguintes:

- desempenho escalável: um arquivo em Zebra pode ser entrelaçado entre múltiplos servidores de armazenamento, permitindo que seu conteúdo seja transferido em paralelo; Assim, a taxa de transferência total pode exceder aquela de um único servidor;
- alta eficiência do servidor: os servidores de armazenamento são mais eficientes quando manipulam grandes transferências de dados pois as pequenas transferências causam grande carga no servidor; Os clientes Zebra utilizam eficientemente os servidores de armazenamento escrevendo neles em grandes transferências, mesmo que as aplicações estejam escrevendo pequenos arquivos;
- grande disponibilidade dos dados: Zebra pode tolerar a perda de qualquer máquina no sistema; Isto é conseguido mantendo a paridade do conteúdo do sistema de arquivos; Se um servidor parar de funcionar seu conteúdo pode ser reconstruído utilizando-se a informação de paridade; O uso da paridade permite prover a disponibilidade de um sistema que mantém cópias redundantes de seus arquivos, a uma fração do custo de armazenamento;
- carregamento uniforme dos servidores: o entrelaçamento de arquivos faz com que a carga imposta por um arquivo muito requisitado seja dividida por todos os servidores que contenham frações daquele arquivo; Em sistemas de arquivos distribuídos tradicionais, tais arquivos afetam o desempenho dos servidores que os contêm, fazendo com que tais arquivos sejam cuidadosamente distribuídos entre os servidores para balancear a carga.

Zebra difere dos sistemas de arquivos distribuídos existentes, no sentido em que o entrelaçamento não é feito a nível de arquivo. Em vez disso, Zebra entrelaça a nível de cliente: todos os novos dados de um cliente são agrupados em uma única cadeia (*stream*) lógica[8], a qual é entrelaçada entre

[8] Independentemente do arquivo ao qual pertençam.

os servidores de armazenamento. Os dados escritos nos servidores em um único passo pelo cliente é chamado faixa (*stripe*). A porção da faixa que é escrita em cada servidor é chamada fragmento de faixa (*stripe fragment*). Os clientes computam a paridade dos fragmentos enquanto estes são escritos. No final da faixa, o cliente escreve a paridade resultante e inicia uma nova computação de paridade.

A Figura 3.8 ilustra os componentes de Zebra.

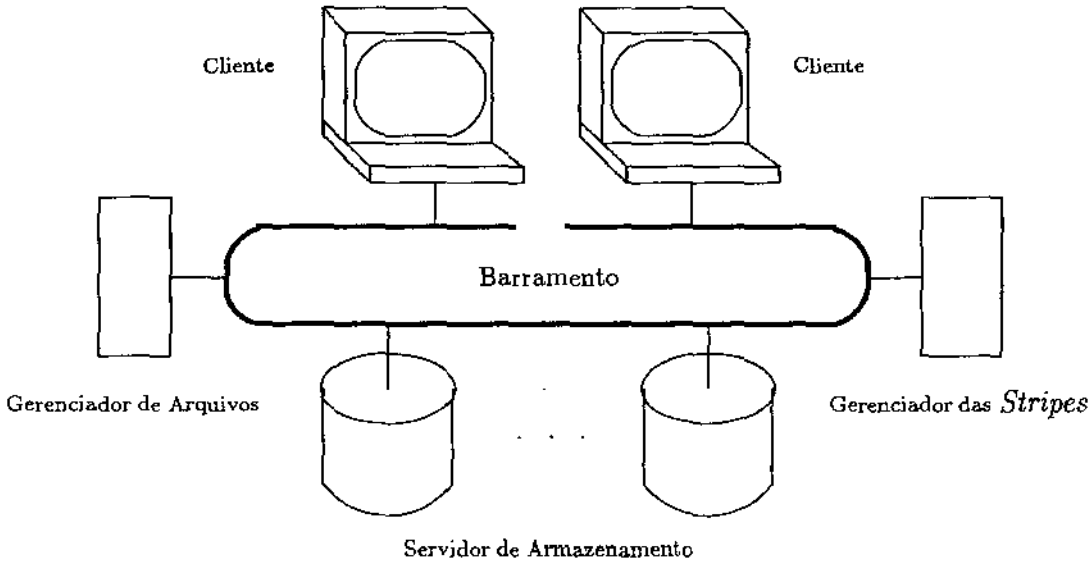

Figura 3.8: A Arquitetura Zebra

Os servidores de armazenamento são apenas repositórios para os fragmentos de faixa. Eles criam novos fragmentos em resposta a requisições dos clientes e os mantêm até que sejam apagados pelo gerenciador de faixas. Os fragmentos são opacos aos servidores de armazenamento, isto é, os servidores não sabem nada à respeito dos arquivos ou blocos que os fragmentos contêm. Esta funcionalidade torna possível implementar o servidor de armazenamento de várias formas diferentes. Uma opção é construir uma máquina especial, otimizada para armazenar fragmentos de faixa. Outra é armazená-los como arquivos locais. Esta última abordagem não é apenas mais fácil de implementar mas também permite que os servidores de armazenamento sejam servidores de arquivos tradicionais.

O gerenciador de arquivos manipula os metadados do sistema, isto é, o espaço de nomes e o mapeamento dos blocos lógicos dos arquivos em fragmentos de faixa. O gerenciador de arquivos é um recurso crítico, pois o sistema de arquivos não pode ser acessado se seus metadados não estiverem disponíveis. Zebra utiliza um gerenciador de arquivos de reserva que pode assumir o lugar do gerenciador principal em caso de falha deste. Durante operação normal, o gerenciador de arquivos armazena todas as alterações nos metadados no servidor reserva. Se o primeiro servidor falhar, o reserva usa essa informação para reconstruir o estado corrente dos metadados.

A centralização do serviço de nomes e o mapeamento de arquivos no gerenciador de arquivos constitui um gargalo de desempenho. Uma técnica para eliminar esse problema é fazer com que os

clientes mantenham um *cache* com tais informações.

O gerenciador de faixas é responsável por controlar o espaço de armazenamento nos servidores de armazenamento. Ele mantém uma lista de todas as faixas do sistema, e quais blocos de arquivos elas contêm. Quando os blocos são apagados ou sobrescritos, a lista é atualizada apropriadamente. Zebra utiliza um gerenciador de faixas de reserva para garantir que seus serviços estejam sempre disponíveis.

A arquitetura Zebra provê alto desempenho no acesso a arquivos grandes ou pequenos. Grandes arquivos são entrelaçados para diminuir o tempo de transferência, e pequenos arquivos são agrupados para reduzir a carga do servidor. O resultado é um sistema de arquivos distribuído custo-efetivo, escalável e confiável, adequado a supercomputação e a processamento de transações.

Capítulo 4

A Arquitetura RAID 5

No capítulo anterior, foram apresentadas diversas técnicas de construção de subsistemas de discos magnéticos que permitem reduzir o tempo de posicionamento, a latência rotacional e o tempo de transferência de dados. Vimos técnicas que extendem as arquiteturas convencionais, bem como novas propostas de arquitetura. Nesta última categoria estava a arquitetura conhecida como RAID 5 (*Redundant Array of Inexpensive Disks - level 5*), objeto desta tese.

Neste capítulo detalhamos esta arquitetura.

4.1 Introdução

Matrizes de disco têm sido tradicionalmente usadas em supercomputadores para se obter altas taxas de transferência de dados. Isto é conseguido através de entrelaçamento de dados e acessos simultâneos ao discos componentes da matriz [Kim 86].

Por hora, vamos supor que a matriz não mantém informações de redundância. Fazemos essa suposição para que o conceito de entrelaçamento fique claro.

A unidade de entrelaçamento, denominada bloco, pode ser um *bit*, um *byte*, um setor, uma página, uma trilha, etc. O i-ésimo bloco lógico é mapeado no bloco físico $\lfloor i/N \rfloor$ do disco $i \ mod \ N$. O conjunto dos N discos é tratado como se fosse um único disco de maior capacidade e mais rápido. Leituras e escritas no grupo de N blocos $\{D_{Ni}, D_{Ni+1}, \ldots, D_{N(i+1)-1}\}$ podem ser feitas em paralelo usando-se uma única rotação dos discos. Se a leitura não estiver alinhada em uma fronteira de N blocos, ou se envolver mais de N trilhas, então múltiplas rotações serão necessárias para se completar a leitura.

Podemos ver, esquematicamente, um exemplo desta arquitetura na Figura 4.1. O espalhamento de dados entre três discos de B blocos cada, dá origem a um disco lógico maior com $3B$ blocos. Uma leitura ou escrita seqüencial dos dados D_0, D_1 e D_2 pode ser executada em paralelo a uma taxa de transferência três vezes maior que a de um único disco.

Para que informações de redundância sejam acrescentadas à matriz, parte de sua capacidade de armazenamento deve ser sacrificada. Este fato é, geralmente, bem aceito pelas organizações que compram o subsistema desde que o fator de utilização final da matriz seja suficientemente alto. Durante muito tempo, o exemplo mais popular de um subsistema de discos tolerante à falhas foi o IBM AS400 [IBM 88], no qual a informação de redundância e o entrelaçamento seguiam a proposta RAID 3 [Pat 88].

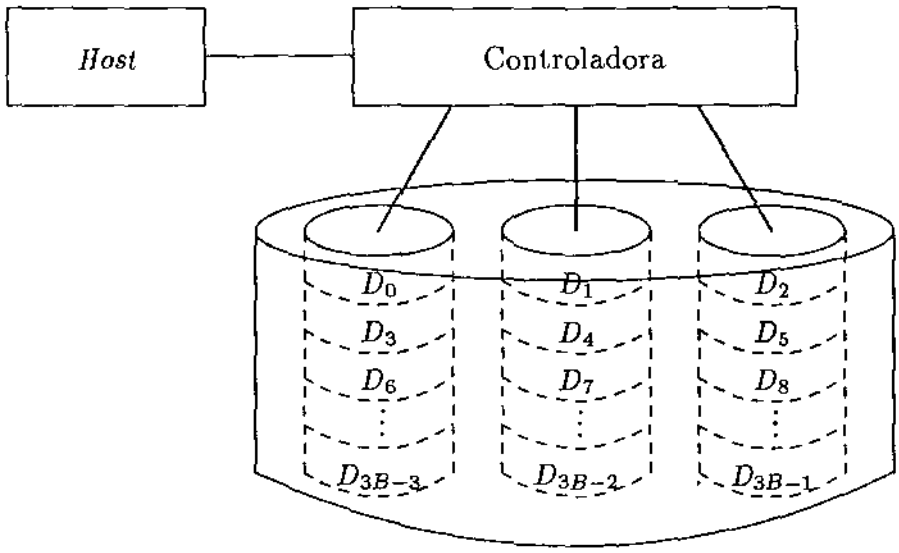

Figura 4.1: Exemplo de Entrelaçamento de Dados, sem Redundância

O posicionamento e o cálculo da informação de redundância são fatores decisivos no desempenho deste tipo de arquitetura (RAID). A Subseção 3.3.2 apresentou, de maneira genérica, como os diversos níveis da arquitetura RAID determinavam o posicionamento e o cálculo da redundância.

A próxima seção apresenta as diversas possibilidades de posicionamento dos dados e da informação de redundância na arquitetura RAID 5, ou seja, as diversas possibilidades do mapeamento lógico para físico de um disco RAID 5. A Seção 4.3 apresenta como a informação de redundância é calculada.

4.2 Mapeamento Lógico para Físico de um Disco RAID 5

Inicialmente, consideremos as seguintes definições:

- **bloco** (*block*): é a mínima unidade de transferência de um dispositivo RAID 5;
- **unidade de faixa** (*stripe unit*): é a unidade de entrelaçamento de dados; Isto é, o grupo de blocos logicamente contíguos que são posicionados consecutivamente em um único disco, antes que se coloquem blocos em outro disco;
- **faixa de redundância** (*redundance stripe*): é a mínima coleção de unidades de faixa sobre o qual a redundância é computada; Nesta seção, o termo faixa, quando não qualificado, se refere a faixa de redundância;
- **linha** (*row*): é a mínima coleção de discos sobre a qual uma faixa de redundância pode ser posicionada;

- **coluna** (*column*): é uma coleção de discos formada pela escolha de um disco em cada linha; Um disco não pode pertencer a mais de uma coluna.

A unidade de faixa é um dos parâmetros mais importantes na configuração de um dispositivo RAID 5. Ela afeta diretamente o desempenho das requisições moderadas[1]. Requisições pequenas [2] não são afetadas pois cabem inteiramente em uma unidade de faixa. Requisições Grandes[3] são pouco sensíveis à unidade de faixa, pois acessos a unidades de faixa fisicamente contígüas podem ser feitas de uma só vez por um *software* mais elaborado. Por exemplo, a Figura 4.2 mostra duas requisições que poderiam ser feitas de uma só vez.

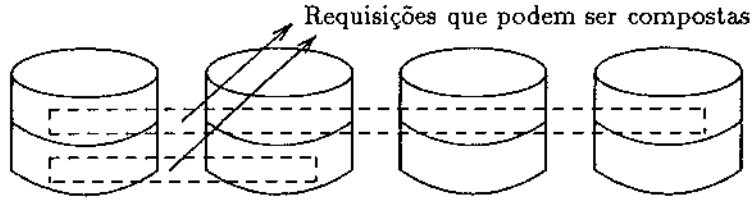

Figura 4.2: Exemplo de Composição de Requisições

O tamanho da faixa de redundância afeta diretamente o desempenho das requisições moderadas de escritas. Escritas são mais eficientes quando têm o mesmo tamanho da faixa de redundância. Isto porque a nova informação de redundância pode ser calculada sem necessidade de leitura adicional. Assim, é desejável escolher faixas de redundância pequenas, de modo que a maioria das escritas possam ser executadas como escritas de faixas de redundância.

A Figura 4.3 ilustra os conceitos introduzidos nesta seção. Ela ajuda a ver que o endereço lógico (*logical block address*) de um bloco lógico[4] da matriz RAID 5 pode ser visto como uma quádrupla ordenada: (número da linha dentro da matriz, número da faixa dentro da linha, número da unidade de faixa dentro da faixa, número do bloco dentro da unidade de faixa). Simbolicamente: ($RowID$, $StripeID$, $StripeUnitID$, $BlockID$).

Para simplificar as equações de posicionamento dos dados e da redundância, supomos que a unidade de faixa tem o mesmo tamanho que o bloco, o qual é um setor. Assim, o endereço lógico de um bloco RAID 5 fica reduzido a ($RowID$, $StripeID$, $StripeUnitID$).

As seguintes definições e equações apresentam o mapeamento de um bloco lógico em seu endereço lógico:

[1] Requisição moderada é aquela que envolve múltiplas unidades de faixa, mas não envolve cada disco mais que uma vez.

[2] Requisição pequena é aquela que cabe inteiramente em uma unidade de faixa, e não cruza as fronteiras da unidade de faixa.

[3] Requisição grande é aquela que é grande o suficiente a ponto de envolver vários discos mais de uma vez.

[4] Bloco do disco RAID 5 visto pela aplicação. Os blocos lógicos são numerados, em ordem crescente, a partir de 0, até chegar ao limite da capacidade do disco.

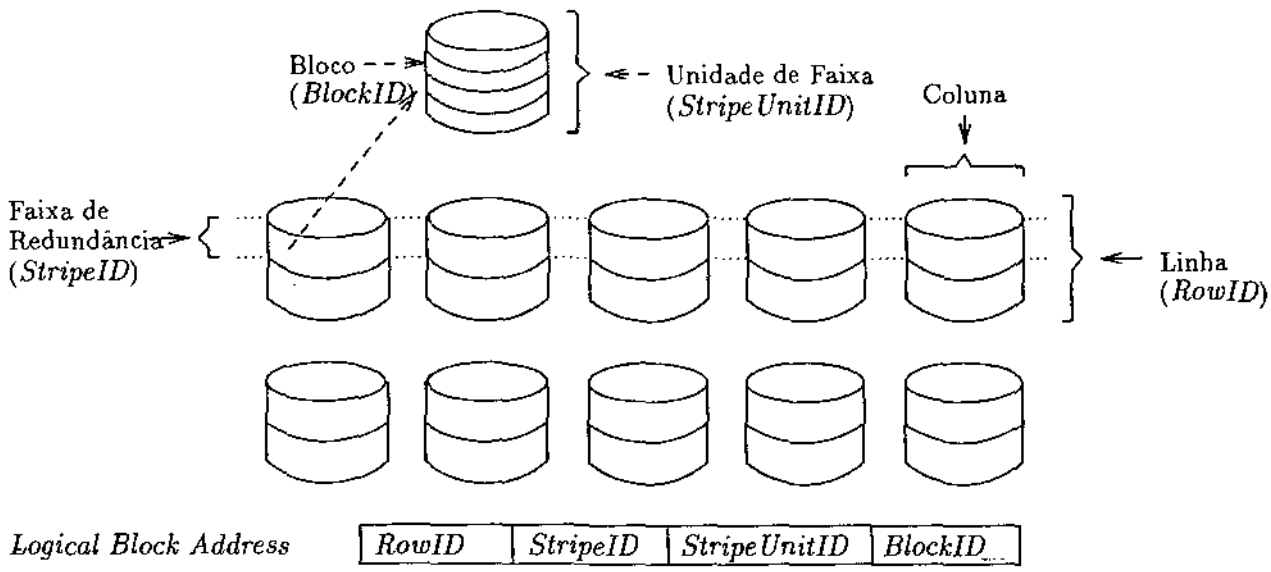

Figura 4.3: Conceitos Empregados no Mapeamento de Blocos Lógicos

$$
\begin{aligned}
n &= \text{número de colunas na matriz} \\
m &= \text{número de linhas da matriz} \\
ndata &= \text{número de unidades de faixa contendo} \\
&\quad \text{dados por faixa de redundância} \\
&= n-1 \\
BlockNumber &= \text{bloco lógico} \\
StripeUnitID &= BlockNumber \bmod ndata \\
RowID &= (BlockNumber \operatorname{div} ndata) \bmod m \\
StripeID &= (BlockNumber \operatorname{div} ndata) \operatorname{div} m \\
LogicalStripeID &= BlockNumber \operatorname{div} ndata
\end{aligned}
$$

4.2.1 Funções de Mapeamento

Formalmente, definimos um mapeamento lógico para físico de uma arquitetura RAID 5 como um par de funções:

1. função de posicionamento dos dados (*data placement function*);

2. função de posicionamento da redundância (*redundance placement function*).

A função de posicionamento dos dados é uma função um-para-um que mapeia um endereço lógico de bloco em um endereço físico de disco (*Disk*,*Sector*).

A função de posicionamento da redundância é uma função muitos-para-um que mapeia um endereço lógico de bloco a um endereço físico de disco. Esta função especifica para cada bloco lógico seu correspondente setor de redundância.

Freqüentemente, é conveniente imaginar uma arquitetura RAID 5 como uma matriz bidimensional de discos e, assim, identificar um disco físico da matriz pelo seu número de linha e coluna. Assim, um endereço físico de disco pode ser representado pela tripla ordenada (*Row*,*Column*,*Sector*).

O número de formas diferentes que a redundância pode ser posicionada relativamente aos dados é muito grande. Nos limitaremos àquelas que satisfazem as seguintes propriedades:

- unidades de faixa que pertencem à mesma faixa de redundância não são mapeadas em uma mesma coluna; Esta propriedade é conhecida como ortogonalidade da arquitetura RAID 5; Garante que a falha em uma única coluna não resulta em indisponibilidade dos dados;

- a redundância de qualquer requisição de escrita que envolva exatamente uma faixa de redundância pode ser calculada sem ter que se ler dados anteriores dos discos.

No que se segue, apresentamos seis formas de mapeamento, propostas em [Lee90] e em [Lee 91], que aderem às propriedades anteriores para o posicionamento da informação de redundância.

Mapeamento Assimétrico-Direito

No mapeamento assimétrico-direito (*right-asymmetric*) os dados são colocados seqüencialmente, da esquerda para a direita, nas faixas de redundância. Para cada faixa de redundância sucessiva, o ponto no qual a unidade de faixa de redundância é inserido é rotacionado uma unidade de faixa para a direita.

Função de Posicionamento dos Dados:

$$\begin{aligned} Row &= RowID \\ Column &= StripeUnitID, \text{ se } StripeID \bmod n > StripeUnitID \\ &= StripeUnitID + 1, \text{ de outra forma} \\ Sector &= StripeID \end{aligned}$$

Função de Posicionamento da Redundância:

$$\begin{aligned} Row &= RowID \\ Column &= StripeID \bmod n \\ Sector &= StripeID \end{aligned}$$

A Figura 4.4 ilustra o mapeamento assimétrico-direito. Para esta e para as demais figuras sobre mapeamento, cada quadrado corresponde a uma unidade de faixa, cada coluna de quadrados corresponde a um disco, cada linha de quadrados corresponde a uma faixa de redundância e cada matriz corresponde a uma linha de discos. É mostrado, apenas, o padrão de repetição.

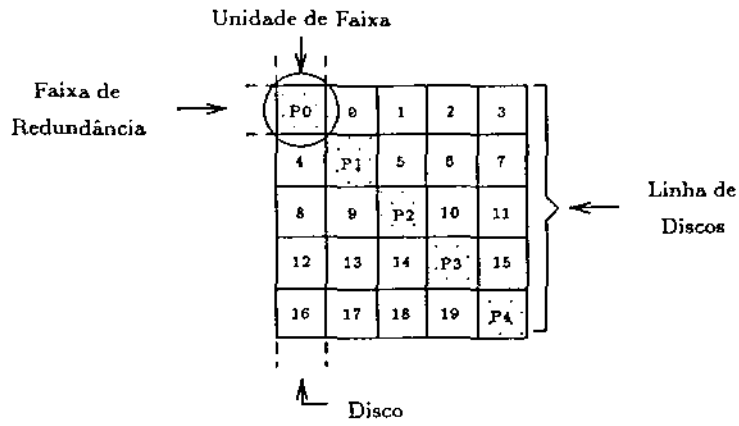

Figura 4.4: Mapeamento Assimétrico-Direito

Mapeamento Assimétrico-Esquerdo

No mapeamento assimétrico-esquerdo (*left-asymmetric*) os dados são colocados seqüencialmente, da esquerda para a direita, nas faixas de redundância. Para cada faixa de paridade sucessiva, o ponto no qual a unidade de faixa de redundância é inserida é rotacionado uma unidade de faixa para a esquerda.

Função de Posicionamento dos Dados:

$$\begin{aligned} Row &= RowID \\ Column &= StripeUnitID, \text{ se } (-StripeID - 1) \bmod n > StripeUnitID \\ &= StripeUnitID + 1, \text{ de outra forma} \\ Sector &= StripeID \end{aligned}$$

Função de Posicionamento da Redundância:

$$\begin{aligned} Row &= RowID \\ Column &= (-StripeID - 1) \bmod n \\ Sector &= StripeID \end{aligned}$$

A Figura 4.5 ilustra o mapeamento assimétrico-esquerdo.

Mapeamento Simétrico-Direito

O mapeamento simétrico-direito (*right-symmetric*) é derivado do mapeamento da arquitetura RAID 4, mediante rotações à direita das faixas de redundância. Na Figura 4.6 apresentamos, informalmente, o mapeamento da arquitetura RAID 4.

0	1	2	3	P0
4	5	6	P1	7
8	9	P2	10	11
12	P3	13	14	15
P4	16	17	18	19

Figura 4.5: Mapeamento Assimétrico-Esquerdo

0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3
16	17	18	19	P4

Figura 4.6: Mapeamento em RAID 4

Função de Posicionamento dos Dados:

$$\begin{aligned} Row &= RowID \\ Column &= (StripeUnitID + StripeID + 1) \bmod n \\ Sector &= StripeID \end{aligned}$$

Função de Posicionamento da Redundância:

$$\begin{aligned} Row &= RowID \\ Column &= StripeID \bmod n \\ Sector &= StripeID \end{aligned}$$

A Figura 4.7 ilustra o mapeamento simétrico-direito.

P0	0	1	2	3
7	P1	4	5	6
10	11	P2	8	9
13	14	15	P3	12
16	17	18	19	P4

Figura 4.7: Mapeamento Simétrico-Direito

Mapeamento Simétrico-Esquerdo

O mapeamento simétrico-esquerdo (*left-symmetric*) é derivado do mapeamento da arquitetura RAID 4, mediante rotações à esquerda das faixas de redundância.

Função de Posicionamento dos Dados:

$$\begin{aligned} Row &= RowID \\ Column &= (StripeUnitID - StripeID) \bmod n \\ Sector &= StripeID \end{aligned}$$

Função de Posicionamento da Redundância:

$$\begin{aligned} Row &= RowID \\ Column &= (-StripeID - 1) \bmod n \\ Sector &= StripeID \end{aligned}$$

A Figura 4.8 ilustra o mapeamento simétrico-esquerdo.

0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figura 4.8: Mapeamento Simétrico-Esquerdo

Mapeamento Simétrico-Esquerdo-Extendido

O mapeamento simétrico-esquerdo-extendido (*extended-left-symmetric*) é derivado de uma matriz sem redundância, deslocando-se verticalmente uma unidade de faixa de dados ao se inserir uma unidade de faixa de redundância. Para cada faixa de redundância sucessiva, o ponto no qual a unidade de faixa de redundância é inserida é rotacionado de uma unidade de faixa para a esquerda. Em matrizes com apenas uma linha de discos, este mapeamento é idêntico ao simétrico-esquerdo.

A seguir, utilizamos o posicionamento dos dados numa matriz sem redundância para especificar as funções de posicionamento dos dados e de posicionamento da redundância no mapeamento simétrico-esquerdo-extendido. O posicionamento dos dados numa matriz sem redundância é definido por:

$$\begin{aligned} Row' &= (BlockNumber \text{ div } n) \bmod m \\ Column' &= BlockNumber \bmod n \\ Sector' &= (BlockNumber \text{ div } n) \text{ div } m \end{aligned}$$

Função de Posicionamento dos Dados:

$$\begin{aligned} Row &= Row' \\ Column &= Column' \\ ColAdj &= 1 \text{ se } \exists\ i \text{ tal que } 0 \le i \le Sector' \bmod (n-1) \text{ e} \\ &\quad (-m \times i - Row' - 1) \bmod n = Column' \\ &= 0, \text{ de outra forma} \\ Sector &= Sector' + Sector' \text{ div } ndata + ColAdj \end{aligned}$$

Função de Posicionamento da Redundância:

$$\begin{aligned} Row &= RowID \\ Column &= (-LogicalStripeID - 1) \bmod n \\ Sector &= StripeID \end{aligned}$$

A Figura 4.9 ilustra o mapeamento simétrico-esquerdo-extendido.

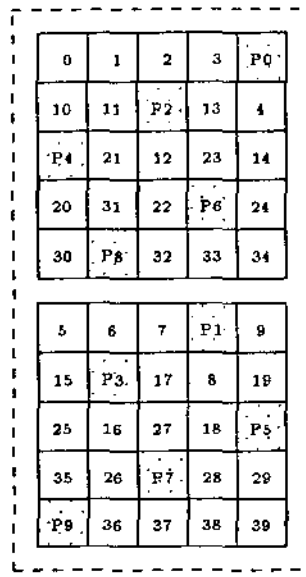

Figura 4.9: Mapeamento Simétrico-Esquerdo-Extendido

Mapeamento Simétrico-Esquerdo-Plano

O mapeamento simétrico-esquerdo-plano (*flat-left-symmetric*) é derivado do mapeamento simétrico-esquerdo-extendido, agrupando-se todas as redundâncias e as colocando em deslocamentos idênticos dentro de cada disco.

A seguir, utilizamos o posicionamento dos dados numa matriz sem redundância para especificar as funções de posicionamento dos dados e de posicionamento da redundância do mapeamento simétrico-esquerdo-plano. O posicionamento dos dados numa matriz sem redundância foi definido anteriormente.

Função de Posicionamento dos Dados:

$$\begin{aligned} Row &= Row' \\ Column &= Column' \\ Sector &= Sector' + Sector' \text{ div } ndata \end{aligned}$$

Função de Posicionamento da Redundância:

$$\begin{aligned} Row &= RowID \\ Column &= (-LogicalStripeID - 1) \bmod n \\ Sector &= (StripeID \text{ div } n) \times n + n - 1 \end{aligned}$$

A Figura 4.10 ilustra o mapeamento simétrico-esquerdo-plano.

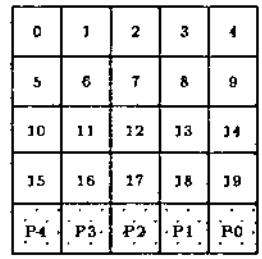

Figura 4.10: Mapeamento Simétrico-Esquerdo-Plano

4.2.2 Desempenho dos Mapeamentos

Em [Lee90] foram conduzidos vários experimentos comparativos sobre o desempenho dos diversos tipos de mapeamento, apresentados na seção anterior. Os resultados principais são:

- o desempenho das leituras e escritas pequenas não é sensível ao mapeamento;
- se o desempenho das leituras grandes a uma baixa taxa de requisições for muito importante, então o mapeamento simétrico-esquerdo-plano é o mais adequado;
- se o desempenho das escritas grandes a uma baixa taxa de requisições for muito importante, então o mapeamento simétrico-esquerdo é o mais adequado;
- a altas taxas de requisições, todos os mapeamentos são equivalentes.

4.3 Cálculo da Redundância e Recuperação de Erros

A seção anterior apresentou diversas topologias para o posicionamento dos dados e da informação de redundância. Naquela seção, não era pertinente como a informação de redundância deveria ser calculada, nem como o subsistema de discos seria capaz de se recuperar de uma possível falha num dos elementos da matriz. De fato, existem muitos métodos possíveis de serem aplicados. Na prática, dá-se preferência àqueles que são simples e eficientes.

Esta seção apresenta um desses métodos.

4.3.1 Cálculo da Informação de Redundância

Em qualquer instante, a informação de redundância presente na matriz RAID 5 deve ser capaz de propiciar confiabilidade, isto é, deve poder prover meios de recuperação de erros. Isto significa que a cada operação de escrita na matriz, esta informação deve ser atualizada, de modo que esta premissa continue válida.

É fácil ver que toda operação de escrita na matriz RAID 5 pode ser conceitualmente dividida em operações de escrita em faixas de redundância. Desta forma, é suficiente mostrar como calcular a nova informação de redundância quando uma operação de escrita envolver, no máximo, toda uma faixa de redundância.

O cálculo é muito simples. Faz-se um XOR de todas as unidades de faixa que deverão ser atualizadas na matriz, juntamente com a unidade de faixa de redundância associada. Ou seja, se supusermos (sem perda de generalidade) que S_1, S_2, ..., S_n é um subconjunto a ser atualizado de uma determinada faixa de redundância cuja informação de redundância atual seja R, então a nova informação de redundância é dada por:

$$R = R_{anterior} \bigoplus S_1 \bigoplus S_2 \bigoplus \cdots \bigoplus S_n \bigoplus S_{1,anterior} \bigoplus S_{2,anterior} \bigoplus \cdots \bigoplus S_{n,anterior}.$$

Por exemplo, na Figura 4.11 temos a seguinte situação: existe uma faixa de redundância composta de três unidades de faixa, das quais duas são de dados; o *host* deseja atualizar a unidade de faixa rotulada por D_4. Inicialmente, devem ser lidos o conteúdo atual da unidade de faixa D_4 e da unidade de redundância associada (P_2). Isto pode ser feito facilmente se as rotações dos discos estiverem sincronizadas. Se não estiverem, as leituras devem ser bufferizadas. Calcula-se então a nova informação de redundância através da seguinte expressão: $P_2 = P_{2,anterior} \oplus D_4 \oplus D_{4,anterior}$. O novo conteúdo de P_2, bem como o de D_4, pode ser atualizado na rotação seguinte àquela da leitura dos valores anteriores.

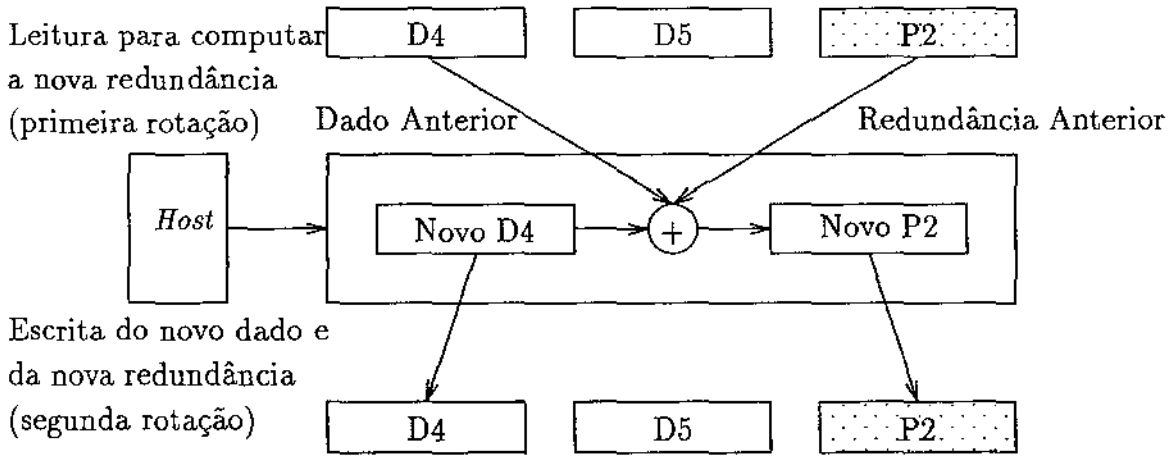

Figura 4.11: Exemplo de Cálculo da Informação de Redundância

4.3.2 Recuperação de Erros

O cálculo da informação de redundância, como descrito na subseção anterior, faz com que o método de recuperação de erros seja essencialmente trivial. Quando uma unidade de faixa lida for considerada defeituosa, esta pode ser reconstruída computando-se o XOR das unidades de faixa restantes, na faixa de paridade, juntamente com o conteúdo da unidade de redundância. Ou seja, se a unidade de faixa S_k de uma faixa de redundância com n discos estiver defeituosa, podemos recuperá-la através da expressão

$$S_k = S_1 \bigoplus \cdots \bigoplus S_{k-1} \bigoplus S_{k+1} \bigoplus \cdots \bigoplus S_n \bigoplus R.$$

Por exemplo, na Figura 4.12 vemos uma faixa de redundância, com quatro discos de dados, na qual o disco 2 está defeituoso. Se for requisitado ao subsistema de discos, dados de uma unidade de faixa que esteja localizada no disco 2, este executa o seguinte algoritmo: o subsistema lê toda a faixa de redundância; Determina cada item da unidade de faixa defeituosa computando o XOR de todos os itens correspondentes das unidades de faixa restantes.

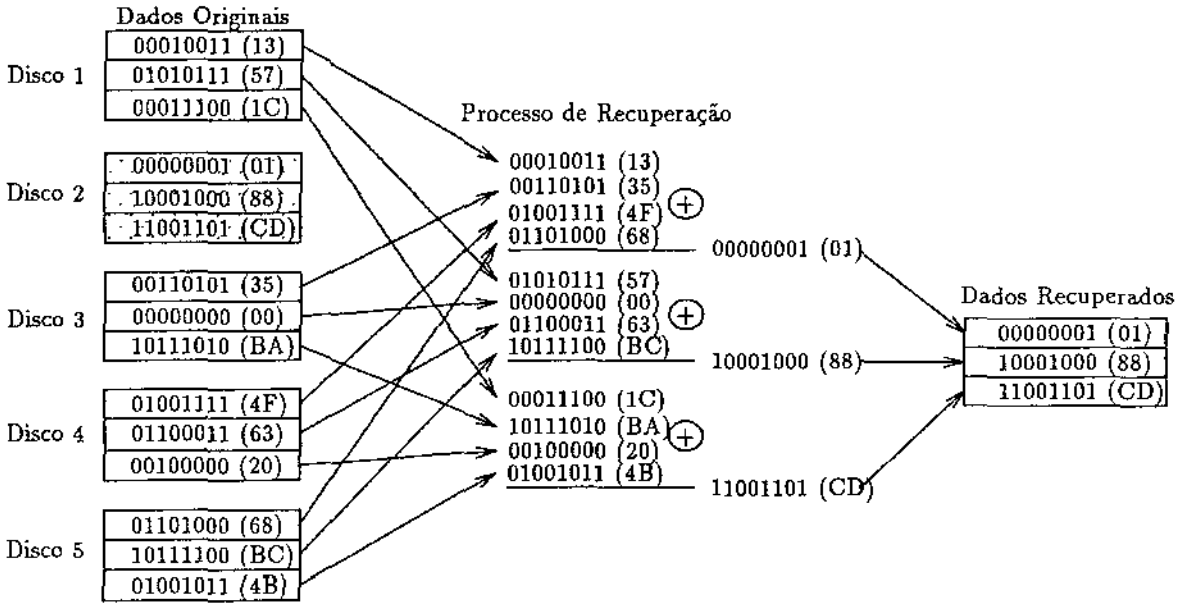

Figura 4.12: Exemplo de Recuperação da Informação

Quando um disco com falha é substituído, seu conteúdo deve ser restaurado no novo disco. Até que este processo esteja concluído, não pode haver outra falha na mesma faixa de redundância, sobre pena de perda de informação. A próxima seção, dentre outras coisas, avalia a confiabilidade da matriz RAID 5, sujeita a esta restrição no processo de recuperação de erros.

4.4 Confiabilidade da Arquitetura RAID 5

Esta seção apresenta diversos modelamentos analíticos a fim de se avaliar a confiabilidade da arquitetura RAID 5. Todas as equações[5] aqui apresentadas, e suas derivações, podem ser encontradas no artigo [Mal 92], o qual é bastante extenso e detalhado.

Durante toda esta seção, vamos supor que a matriz RAID 5 é composta de N grupos de discos[6] (linhas), cada qual formado por D discos de dados e 1 disco de conferência (*check disk*) ($D+1$ colunas), e que o tempo médio para falha, MTTF (*mean time to failure*), de cada disco é exponencialmente distribuído com média $1/\lambda$. Supomos, também, que cada grupo possui seu

[5]Neste trabalho de tese, não nos preocupamos com os detalhes de derivação das equações apresentadas nesta seção. Tais equações são apresentadas por completude e para justificar as conclusões (Seção 4.4.5) desta seção.

[6]Todos do mesmo modelo e do mesmo fabricante

próprio mecanismo de recuperação de erros, possibilitando a existência de múltiplos processos desse tipo simultaneamente. Se não especificado o contrário, supomos que o tempo de recuperação de erros é exponencialmente distribuído com média $1/\mu$.

Definimos confiabilidade dos dados (*data reliability*) como sendo a probabilidade deles estarem íntegros até o tempo t. Um modelo hierárquico de dois níveis para a confiabilidade, conhecido como diagrama de blocos de confiabilidade, RBD (*reliability block diagram*), é mostrado na Figura 4.13. Este modelo possui uma estrutura serial, onde cada bloco representa um grupo de discos. Até que seja explicitamente mencionado o contrário, supomos que os grupos se comportam independentemente uns dos outros.

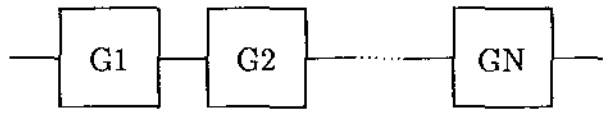

Figura 4.13: Diagrama de Blocos de Confiabilidade

Baseados nessas premissas, temos que a confiabilidade dos dados, conseqüentemente da matriz de discos, pode ser determinada através da equação

$$R_{da}(t) = \prod_{i=1}^{N} R_i(t), \tag{4.1}$$

onde $R_i(t)$ representa a confiabilidade do grupo i, computada utilizando-se o modelo de Markov mostrado na Figura 4.14.

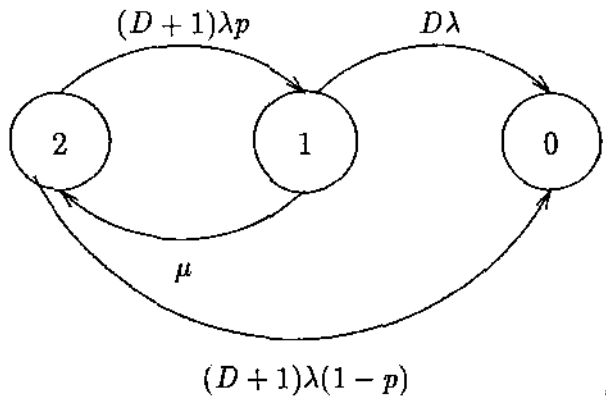

Figura 4.14: Modelo de Markov para a Confiabilidade de um Grupo

No Modelo de Markov da Figura 4.14, cada estado tem uma interpretação bem definida. No estado 2, todos os discos estão operacionais. Após a falha de algum disco, o estado do sistema

muda de 2 para 1 e o processo de recuperação de erros é iniciado. Contudo, a matriz continua funcionando pois os dados estão disponíveis. Se algum outro disco falhar antes que a recuperação esteja completa, os dados estarão perdidos e a matriz será considerada com falha. O estado 0 é o estado de falha do grupo.

Uma observação importante é que o Modelo de Markov desenvolvido permite uma cobertura parcial das falhas, com probabilidade p. Contudo, uma falha não coberta em um grupo causa perda de dados.

Para que a confiabilidade dos dados da matriz (Equação 4.1) esteja plenamente definida, precisamos saber avaliar a confiabilidade de um grupo i. Isto pode ser feito através da equação

$$R_i(t) = A_1 e^{\beta_1 t} + A_2 e^{\beta_2 t}, \tag{4.2}$$

onde

$$\begin{aligned}
\beta_1 &= \frac{-(\mu + (2D+1)\lambda) + \sqrt{\lambda^2 + \mu^2 + \lambda\mu((4D+4)p - 2)}}{2}, \\
\beta_2 &= \frac{-(\mu + (2D+1)\lambda) - \sqrt{\lambda^2 + \mu^2 + \lambda\mu((4D+4)p - 2)}}{2}, \\
A_1 &= \frac{((D+1)(1+p) - 1)\lambda + \mu + \beta_1}{\beta 1 - \beta_2}, \\
A_2 &= \frac{((D+1)(1+p) - 1)\lambda + \mu + \beta_2}{\beta 2 - \beta_1}.
\end{aligned} \tag{4.3}$$

Outro parâmetro importante na avaliação de desempenho da matriz RAID 5 é o tempo médio para perda de dados, MTDL (*mean time to data loss*). No caso de se avaliar a matriz como um todo, temos que

$$MTDL_{da} = \int_0^\infty R_{da}(t)dt = \sum_{j=0}^{N} \frac{\binom{N}{j} A_1^j A_2^{N-j}}{\beta_1 j + \beta_2 (N-j)}. \tag{4.4}$$

Já, quando se olha individualmente para um dado grupo i de discos, temos que

$$MTDL_i = \frac{\mu + ((D+1)(1+p) - 1)\lambda}{(D+1)\lambda(\mu(1-p) + D\lambda)}. \tag{4.5}$$

4.4.1 Falhas de disco Preditivas

Algumas controladoras de disco possuem a habilidade de predizer corretamente certos tipos de falhas antes que elas ocorram. Modificaremos o modelo anterior para levar em consideração essa característica.

Supomos que nenhuma perda de dados ocorre se a controladora prediz corretamente uma falha iminente, e que um disco sobressalente é eletronicamente chaveado e os dados reconstruídos nele antes que o disco com falha seja removido. Esta seqüência de operações não resulta em mudança de estado do sistema.

A controladora pode não ser sempre capaz de predizer uma falha, especialmente aquelas não cobertas. Existe também a possibilidade de falsos alarmes quando a controladora prediz erroneamente uma falha de disco. O tempo para o próximo alarme falso é suposto ser exponencialmente

distribuído com taxa γ. Alarmes falsos são tratados como falhas corretamente preditas e não resultam em mudança de estado do sistema. Isto porque supomos um número ilimitado de discos sobressalentes. Também, com probabilidade $(1-\alpha)$, uma falha iminente devido a uma falha coberta não é preditível. O modelo de Markov, baseado nessas premissas, para cada grupo de discos, é mostrado na Figura 4.15.

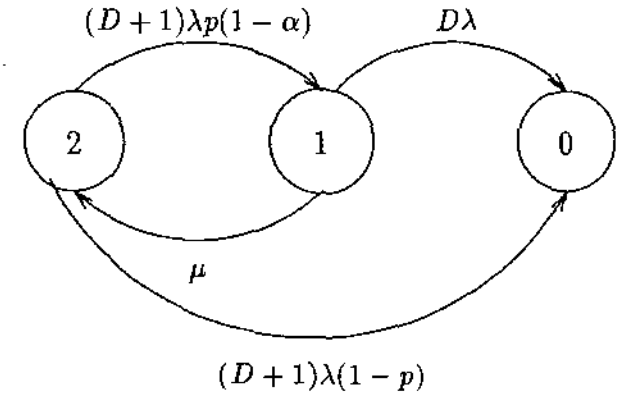

Figura 4.15: Modelo de Markov para a Confiabilidade de um Grupo com Falhas Preditíveis

A confiabilidade de cada grupo tem a mesma forma da Equação 4.2, onde

$$\begin{aligned}
\beta_1 &= \frac{-(((D+1)(2-p\alpha)-1)\lambda+\mu)+\sqrt{X}}{2}, \\
\beta_2 &= \frac{-(((D+1)(2-p\alpha)-1)\lambda+\mu)-\sqrt{X}}{2}, \\
A_1 &= \frac{\beta_1+\mu+D\lambda}{\beta_1-\beta_2}, \\
A_2 &= \frac{\beta_2+\mu+D\lambda}{\beta_2-\beta_1}, \\
X &= ((D+1)((D+1)p^2\alpha^2-2p\alpha)+1)\lambda^2+\mu^2+2((D+1)p(2-\alpha)-1)\lambda\mu. \qquad (4.6)
\end{aligned}$$

O tempo médio para perda de dados em um grupo é dado por

$$MTDL_i = \frac{\mu+((D+1)(1+p(1-\alpha))-1)\lambda}{(D+1)\lambda(\mu(1-p)+D(1-\alpha)\lambda)}. \qquad (4.7)$$

O diagrama de blocos de confiabilidade para a matriz é o mesmo que o da Figura 4.13. A confiabilidade da matriz é computada pela Equação 4.1 e o tempo médio para perda de dados, pela Equação 4.4.

4.4.2 Tipos de Discos Sobressalentes

Até aqui, supomos um número ilimitado de discos sobressalentes. Isto, porém, não corresponde à realidade prática. Em implementações reais, um número fixo pequeno é que é mantido. Nesta

subseção, avaliaremos o efeito causado pela limitação no número de discos sobressalentes na confiabilidade da matriz RAID 5.

Os discos sobressalentes podem ser mantidos *quentes* ou *frios*. Um disco *quente* pode falhar mesmo que não esteja em uso ativo, já um disco *frio* não falha enquanto não estiver substituindo algum disco da matriz. Um disco *quente* pode ser inserido eletronicamente na matriz após a falha de algum disco e o tempo para executar essa inserção é desprezível. Assim, o tempo de recuperação de erros, neste caso, consiste apenas do tempo de reconstrução dos dados no disco sobressalente. As desvantagens dos discos sobressalentes *quentes* são:

1. o mecanismo de inserção representa um adicional no custo do sistema;

2. esses discos podem falhar mesmo não estando em uso ativo;

3. o mecanismo de inserção pode falhar.

A outra opção é manter discos sobressalentes *frios*. Quando um disco falha, um operador é acionado para instalar um novo disco. Após a instalação, ocorre a reconfiguração da matriz e a reconstrução dos dados. Desta forma, o tempo de reparo é maior que no caso de se ter discos sobressalentes *quentes*. Uma solução melhor seria a combinação dos dois métodos. Poucos discos *quentes* seriam mantidos, enquanto que o resto seriam mantidos *frios*. Cada vez que um disco falhasse, um disco *quente* seria usado e um *frio* seria tornado *quente*.

No caso de se manter discos sobressalentes, surge a questão de quantos tais discos devem ser mantidos. Intuitivamente, é claro que se existe um pequeno número de grupos cada qual contendo um grande número de discos, então o número de discos sobressalentes por grupo deve ser grande. Contudo, se existirem um grande número de grupos cada qual contendo um pequeno número de discos, o número de discos sobressalentes por grupo deve ser pequeno.

Nesta subseção, apresentaremos o modelo de confiabilidade para a arquitetura RAID 5 baseados em premissas distintas. Supomos que cada grupo possui M discos sobressalentes, e que para discos *quentes* o tempo de inserção é desprezível.

Supondo que temos apenas discos sobressalentes frios, o modelo de confiabilidade é apresentado na Figura 4.16, com

$$\begin{aligned} \lambda_1 &= (D+1)\lambda_d p(1-\alpha), \\ \lambda_2 &= (D+1)\lambda_d p\alpha + \gamma, \\ \lambda_3 &= (D+1)\lambda_d(1-p), \\ \lambda_4 &= D\lambda_d, \end{aligned} \tag{4.8}$$

sendo α e γ definidos na subseção 4.4.1.

Um estado é uma dupla (i,j) onde i é o numero de discos ativos e j é o número de discos sobressalentes restantes. O estado A é o estado de falhas. Na Figura 4.16, λ_d representa a taxa de falhas de um disco ativo em uso, λ_1 falhas cobertas de um dado disco, λ_2 instalação de um disco sobressalente frio antes que o disco com falha seja substituído, λ_3 falhas não cobertas, e λ_4 falhas de um disco durante a recuperação de erros. μ_3 é a taxa de reconstrução dos dados supondo-se que pelo menos um disco sobressalente esteja disponível. No estado $(G,0)$ não há mais discos sobressalentes. Se um disco falhar neste estados, a taxa de reconstrução dos dados é μ_2 onde $\mu_2 < \mu_3$ devido ao aumento no tempo de reconstrução. Como um disco sobressalente é instalado após a eliminação do

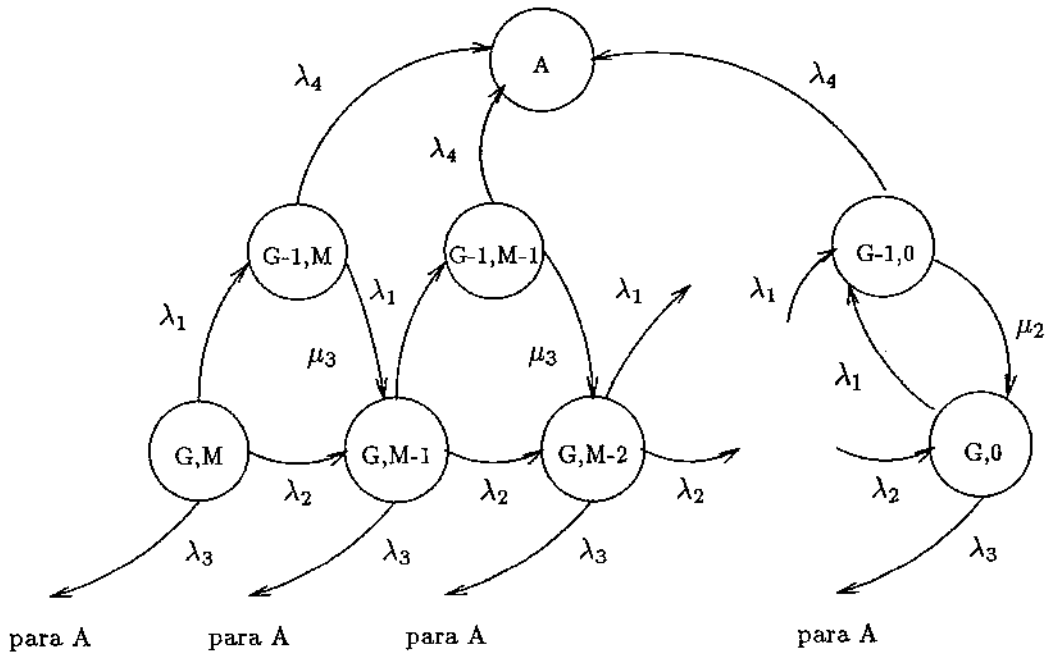

Figura 4.16: Modelo de Markov para a Confiabilidade de um Grupo com Falhas Preditíveis e M Discos Sobressalentes *Frios*

disco com falha, as transições resultam apenas em uma mudança de estado refletindo a diminuição do número de discos sobressalentes em uma unidade.

Consideremos, agora, o modelo de confiabilidade que leva em consideração o uso de discos sobressalentes *quentes*. Este modelo, visto na Figura 4.17, difere do anterior pois, aqui, existem transições dos estados $(G-1, M-i+1)$ para os estados $(G-1, M-i)$, $1 < i < M$, significando a falha dos discos sobressalentes *quentes*. Há também uma mudança na taxa de transições entre os estados $(G, M-i+1)$ e $(G, M-i)$, $1 < i < M$, onde $\lambda_{s,i} = i\lambda_{sp} + (D+1)\lambda_d p\alpha + \gamma$, com λ_{sp} sendo a taxa de falhas de um disco sobressalente *quente* e $\lambda_{sp} < \lambda_d$. A taxa de recuperação de erros enquanto pelo menos um disco sobressalente *quente* estiver disponível vale μ_1 e $\mu_2 < \mu_1$.

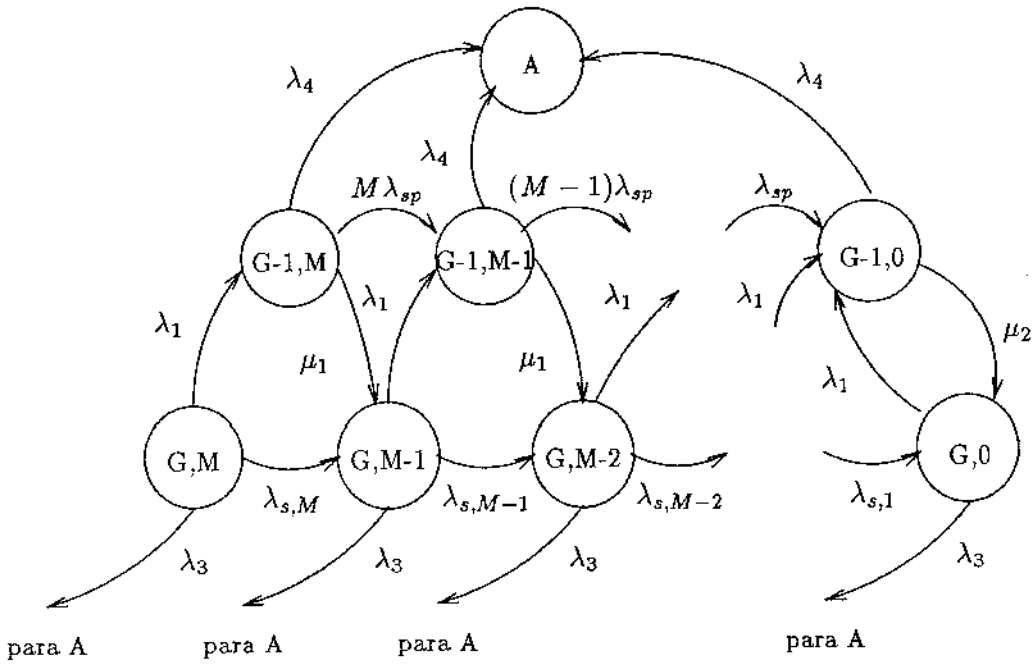

Figura 4.17: Modelo de Markov para a Confiabilidade de um Grupo com Falhas Preditíveis e M Discos Sobressalentes *Quentes*

4.4.3 Modelo de Confiabilidade Incluindo o Hardware de Suporte

Uma matriz de discos possui muitos componentes de *hardware*. Estes incluem o adaptador de barramento (*host bus adapter* - HBA), a controladora da matriz (*disk array controller* - DC), as controladoras dos discos (*hard disk controller* - HDD), o circuito de correção de erros (*error correction circuitry* - ECC), o *hardware* de resfriamento (*cooling fan* - CF), as fontes de alimentação (*power supply* - PS), etc. O circuito de correção de erros juntamente com um *cache* de setores estão normalmente residentes em uma única controladora (*single board controller* - SBC).

Até este ponto, consideramos apenas o efeito dos discos magnéticos no modelo de confiabilidade da matriz. Contudo, falhas no *hardware* de suporte pode resultar em perda de informação.

Nesta subseção modelaremos o efeito causado pelo *hardware* de suporte na confiabilidade da matriz de discos, segundo dois tipos de organização de *hardware*.

Posicionamento Serial do Hardware de Suporte

Esta é uma organização simples na qual a matriz de discos possui um conjunto de componentes de *hardware*[7] associado, posicionados serialmente na matriz. Se não houver redundância nesses componentes, uma falha em qualquer um deles resulta em falha da matriz. Se houver redundância, falhas em alguns desses componentes podem ser toleradas. Caso algum componente pare de funcionar, consideramos que a matriz apresenta uma falha de operação. O modelo de confiabilidade hierárquico é mostrado na Figura 4.18.

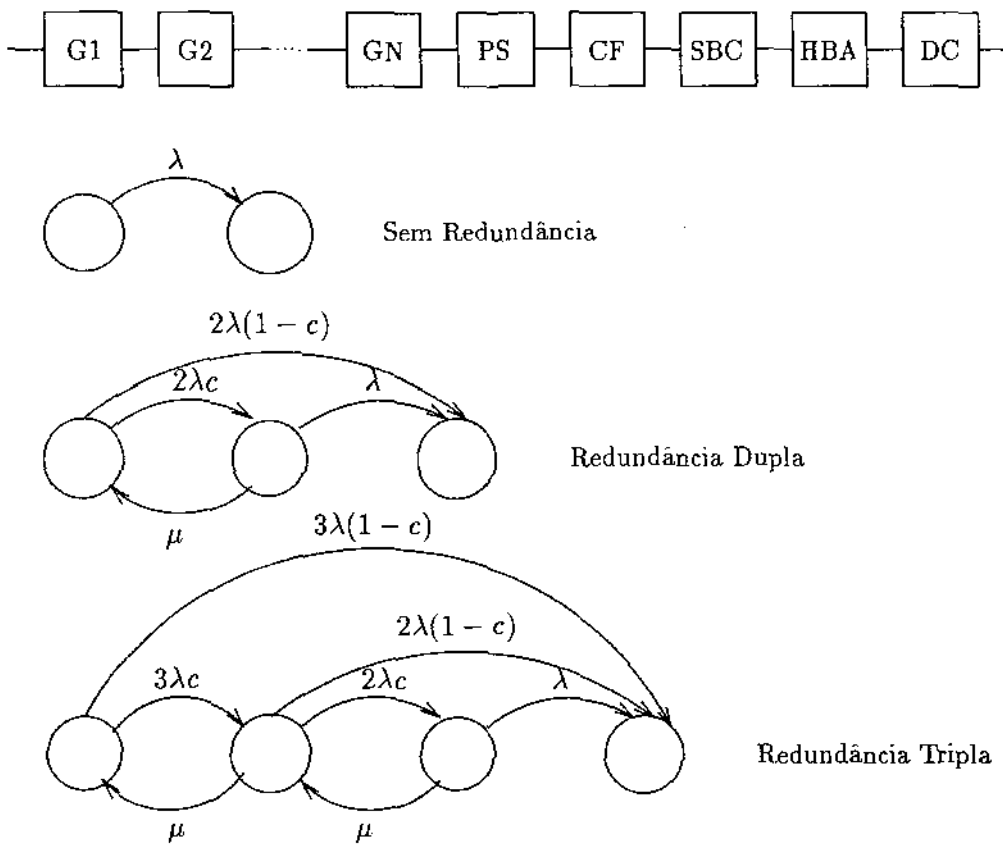

Figura 4.18: Posicionamento Serial do *Hardware* de Suporte

A confiabilidade da matriz é, então, dada por:

$$R_{da}(t) = (\prod_{i=1}^{N} R_i(t))R_{hba}(t)R_{dc}(t)R_{sbc}(t)R_{ps}(t)R_{cf}(t).$$

[7]Por exemplo HBA, fonte de alimentação (*power suply* - PS), ventiladores de resfriamento (*cooling fan* - CF), etc.

Dependendo do número de cópias redundantes de cada componente, o respectivo modelo de confiabilidade é diferente. Mostramos na Figura 4.18 modelos de Markov para componentes sem redundância, com redundância dupla e com redundância tripla. Em cada um desses modelos, λ é a taxa de falhas do componente, μ a taxa de reparos e c a probabilidade de cobertura[8]. Supomos o uso de *hardware* sobressalente *quente*.

Posicionamento Ortogonal do Hardware de Suporte

Schulze [Sch 88] propôs um esquema de posicionamento dos discos e do *hardware* de suporte que torna a matriz de discos mais tolerante à falhas. Os discos são organizados em um reticulado bidimensional, com cada linha representando uma faixa de redundância. *Hardware* de suporte é provido para cada coluna de discos, formando um grupo de suporte de *hardware*. Esta organização, mostrada na Figura 4.19, suporta falhas simples no *hardware* de suporte.

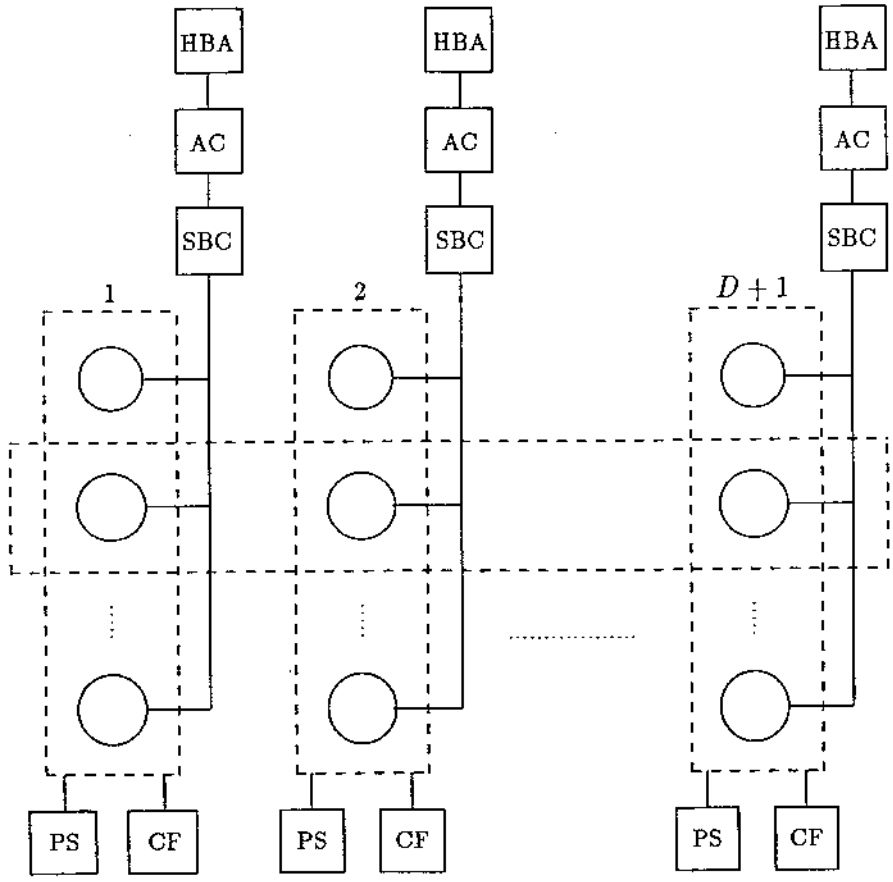

Figura 4.19: Posicionamento Ortogonal do *Hardware* de Suporte

Devido à dependência complexa entre falhas e recuperação de erros, um modelo de confiabilidade

[8] probabilidade de se detectar falhas

simples não é possível de ser conseguido. Modelagem através de uma rede de Petri estocástica resulta em uma cadeia de Markov bastante complexa. Contudo, sua simetria permite desenvolver um modelo menor, bastante aproximado, com apenas quatro estados, mostrado na Figura 4.20.

Supomos que os componentes de *hardware* de suporte são estatisticamente independentes e igualmente constituídos em cada coluna, e que falhas no *hardware* de uma coluna causa a falha de toda a coluna. Supondo uma distribuição exponencial do tempo para falhas de cada componente de suporte de *hardware*, a taxa de falhas de cada coluna λ_{sh} é a soma da taxa de falhas de cada componente e a distribuição do tempo para falhas do *hardware* de cada coluna é exponencial com essa taxa. No estado 2, estado totalmente operacional, o *hardware* de uma coluna pode falhar (transição para o estado 1) sendo reparado a uma taxa μ_{sh}.

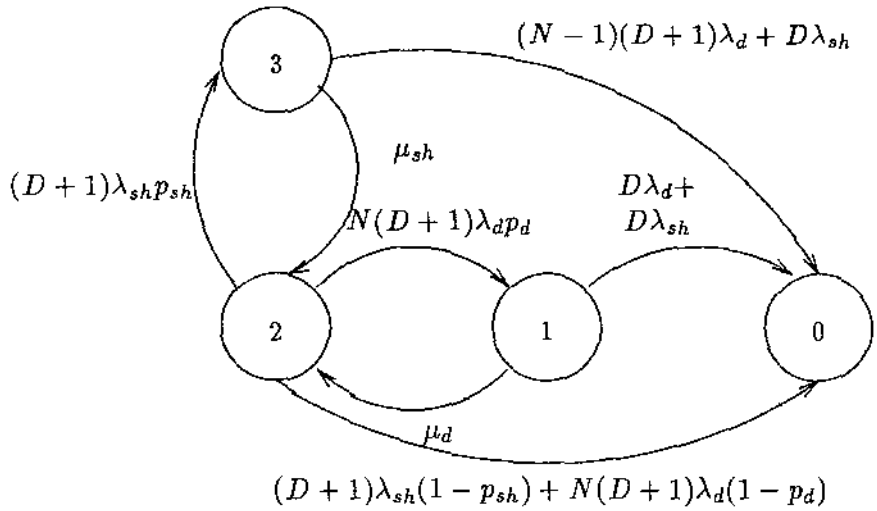

Figura 4.20: Modelo de Confiabilidade Aproximado para o Posicionamento Ortogonal

Uma falha no *hardware* de uma coluna é coberta com probabilidade p_{sh}. Enquanto a operação de reparo estiver em andamento, os discos nessa coluna são considerados não operacionais. Contudo, se $(N-1)(D+1)$ discos falham ou se outro *hardware* de outra coluna falhar antes que o reparo esteja completo, ocorrerá perda de dados (transição para o estado 0). Similarmente, no estado 2, qualquer um dos $N(D+1)$ discos pode falhar (transição para o estado 1) a uma taxa λ_d. Uma falha de disco é coberta com probabilidade p_d e a taxa de reconstrução dos dados é μ_d.

4.4.4 Resultados Numéricos

Em [Mal 92] foram conduzidos vários experimentos a fim de responder a questões fundamentais sobre a viabilidade técnica, prática e econômica de se construir um subsistema de armazenamento RAID 5 em escala comercial. Nesta seção, apresentaremos as conclusões obtidas em [Mal 92].

O modelo RAID 5 básico é uma matriz com 40 discos idênticos, dividida em oito grupos de quatro discos de dados e um de conferência. O tempo para falhas de um disco ativo (de dados e de conferência) é exponencialmente distribuído com média de 40000 horas ($\lambda = 1/40000$ por hora).

A distribuição do tempo para a recuperação de erros é exponencial com taxa μ. Se nenhum disco sobressalente for mantido, então o tempo médio para a recuperação de erros é de 72hs. A taxa de falhas de um disco sobressalente *quente* é $\lambda_{sp} = 1/50000$ por hora. A probabilidade de cobertura vale 0.9.

Em modelos com falhas preditivas, a taxa de alarmes falsos vale $\gamma = 1/100000$ por hora e a probabilidade que uma falha iminente seja corretamente predita vale $\alpha = 0.9$. Para modelos com *hardware* de suporte, MTTF para a fonte de alimentação = 1460 horas, HBA = 123000 horas, cabo de alimentação = 10000000 horas, cabo SCSI = 21000000 horas, equipamento de ventilação = 195000 horas, SBC = 40000 horas e controladora HDD = 30000 horas. O MTTR de qualquer *hardware* de suporte é suposto ser 24 horas ($\mu_{sh} = 1/24$ por hora).

Quão Confiável Deve Ser Cada Disco?

Supomos que discos sobressalentes *quentes* são mantidos e que estão sempre disponíveis quando necessário. A Figura 4.21 mostra como a confiabilidade da arquitetura RAID 5 varia em função da variação do MTTF dos discos constituintes da matriz. O ganho em confiabilidade é significativo quando o MTTF é aumentado de 10000 para 40000 horas. Contudo, o ganho na confiabilidade não aumenta muito se o MTTF dos discos for aumentado para além de 40000 horas.

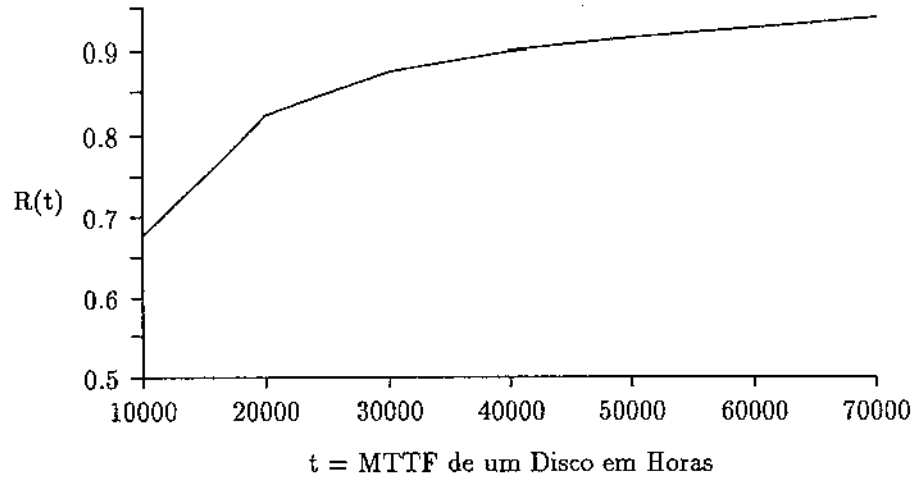

Figura 4.21: Confiabilidade Versus MTTF dos Discos da Matriz em Horas

A Arquitetura RAID 5 é Confiável o Suficiente para Uso em Missões Críticas?

Na Figura 4.22 a confiabilidade da matriz de discos é plotada em função do tempo da missão. RAID 5 é altamente confiável para períodos de operação de 500 horas ou menos.

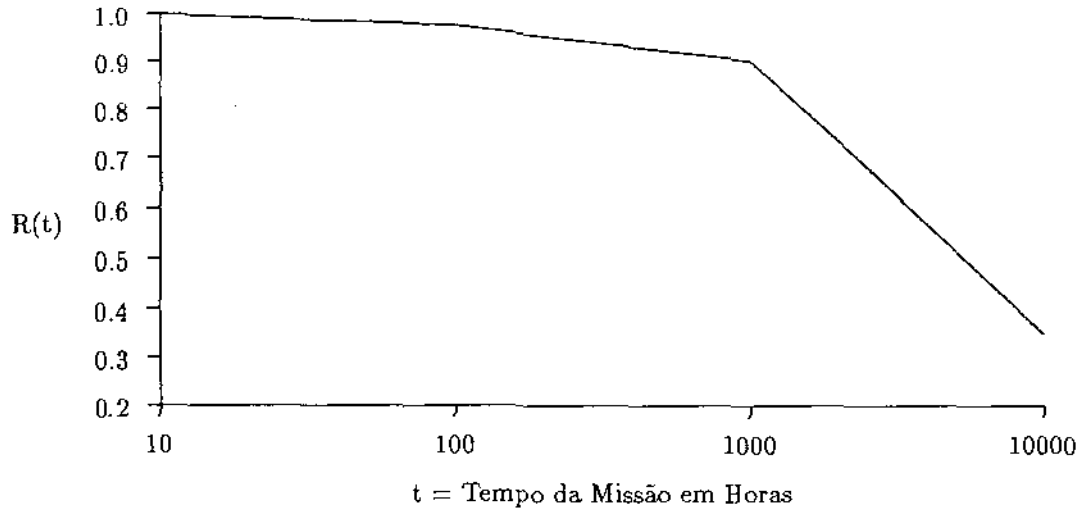

Figura 4.22: Confiabilidade Versus Tempo em Horas

Qual o Efeito do Aumento da Cobertura de Falhas?

Na Figura 4.23, a confiabilidade da matriz de discos (para $t = 1000$ horas) é plotada como função da probabilidade de cobertura. Um aumento tremendo na confiabilidade é conseguido com a melhoria na cobertura das falhas.

Quão Pequeno Deve Ser o Tempo de Reconstrução dos Dados?

Vamos considerar o efeito do tempo médio para reconstrução dos dados (recuperação de erros), MTDR, na confiabilidade da matriz de discos. A Figura 4.24 mostra que o MTDR não afeta significativamente a confiabilidade da matriz. A confiabilidade foi avaliada no tempo $t = 1000$ horas. Variando MTDR de 2 para 100 horas não afeta muito a confiabilidade da matriz. A razão para isto é que o MTTF de um disco é muito maior que seu MTDR. A virtual independência da confiabilidade da matriz em relação ao tempo de reconstrução dos dados sugere o uso de discos sobressalentes *frios* ao invés de *quentes*. Esses últimos estão tão sujeitos à falhas como os discos de dados (os discos *frios* não falham). Além disso, é mais caro manter discos *quentes* do que discos frios.

Quantos Discos Sobressalentes São Necessários?

Avaliaremos, agora, a dependência da confiabilidade da matriz em relação ao número de discos sobressalentes e a seus tipos.

Nos modelos da Figura 4.16 e da Figura 4.17, escolhemos $\mu_3 = 1/50$ por hora (tempo de reconstrução dos dados quando um disco sobressalente *frio* está disponível), $\mu_2 = 1/74$ por hora (tempo de reconstrução dos dados quando nenhum disco sobressalente está disponível) e $\mu_1 = 1/2$ por hora (tempo de reconstrução dos dados quando um disco sobressalente *quente* está disponível).

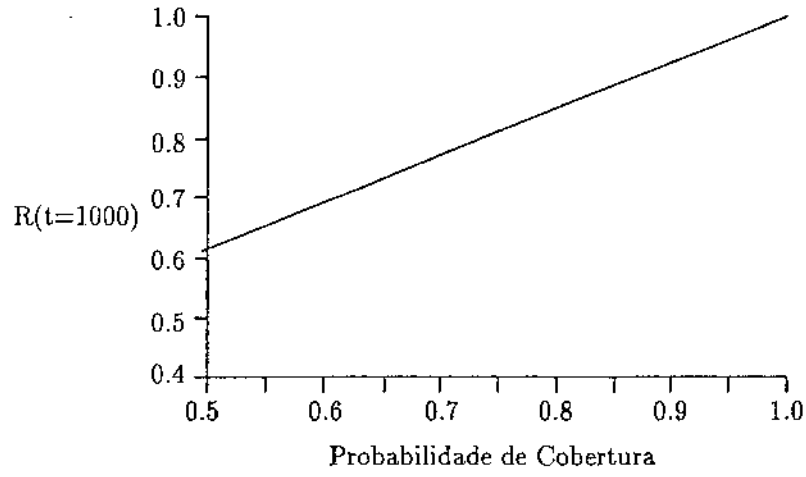

Figura 4.23: Confiabilidade Versus Cobertura

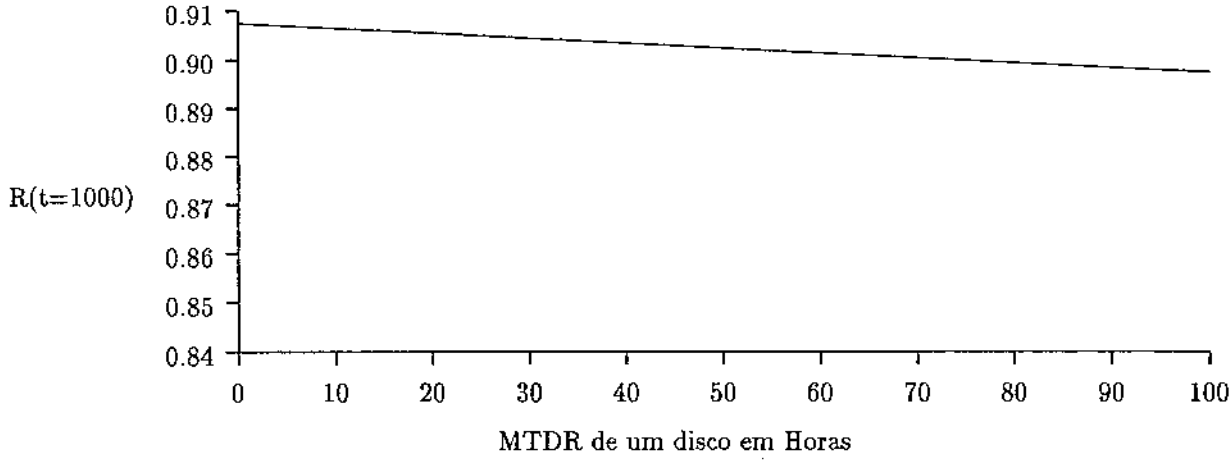

Figura 4.24: Confiabilidade Versus Tempo Médio para Reconstrução dos Dados

O MTTF de um disco sobressalente *quente* é suposto como sendo 50000 horas ($\lambda_{sp} = 1/50000$ por hora), o qual é maior que o MTTF de um disco ativo (40000 horas).

O MTDL para uma matriz de discos como função do número de discos sobressalentes (*frios* e *quentes*) está plotado na Figura 4.25. O ganho no MTDL não é significativo quando o número de discos sobressalentes ultrapassa a dois por grupo.

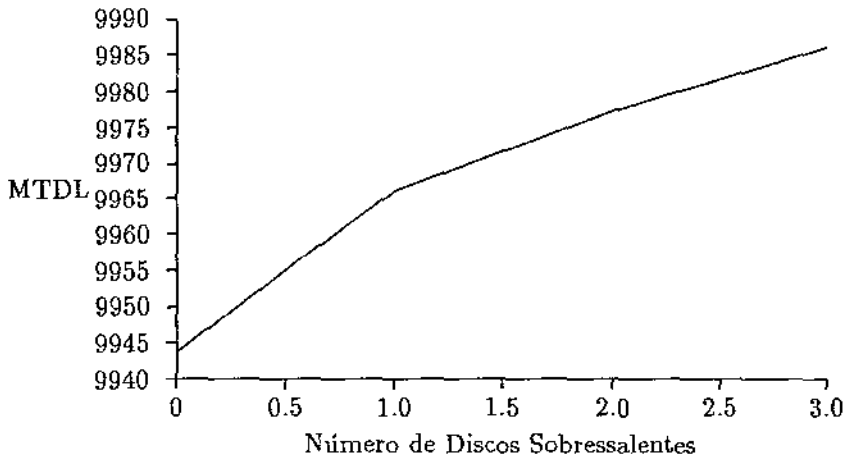

Figura 4.25: MTDL Versus Número de Discos Sobressalentes

A Confiabilidade da Arquitetura RAID 5 é Escalável?

Um passo natural em direção ao aumento de paralelismo nas transferências de I/O seria o aumento das duas dimensões da matriz de discos, aumentando-se o número de discos em um grupo e o número de grupos. Desejamos verificar como se comporta a confiabilidade da matriz face a esse aumento nas dimensões.

Dado um número fixo de discos em um grupo ($D = 8$), o número de grupos (N) é variado. A Figura 4.26 ilustra como a confiabilidade decresce com o aumento do número de grupos. Dado um número fixo de grupos ($N = 8$), o número de discos no grupo (D) é variado. A Figura 4.27 ilustra como a confiabilidade decresce com o aumento em D. Em ambos os casos, a confiabilidade foi avaliada em $t = 1000$ horas. As Figuras 4.26 e 4.27 revelam que a confiabilidade da Arquitetura RAID 5 cai a níveis inaceitáveis com o crescimento nas dimensões da matriz.

4.4.5 Conclusões

Após a análise dos resultados da subseção anterior, chega-se às conclusões que serão citadas a seguir.

Se a matriz de discos for empregada em uma missão crítica de longo tempo, esta deve possuir alta confiabilidade durante um período de tempo muito grande. Os resultados mostraram que a arquitetura RAID 5 não apresenta os requerimentos de confiabilidade em suas implementações atuais. Contudo, a introdução de redundância adicional para os componentes chave pode ajudar a

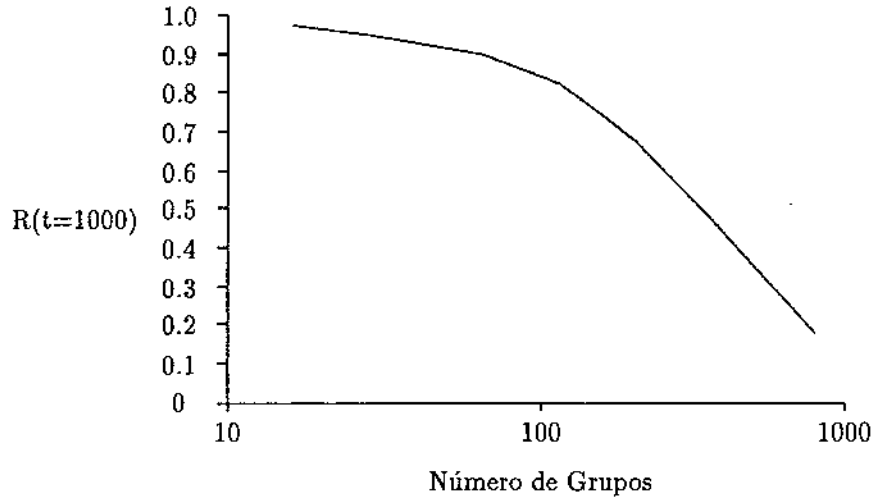

Figura 4.26: Confiabilidade Versus Capacidade de Armazenamento ($D = 8$)

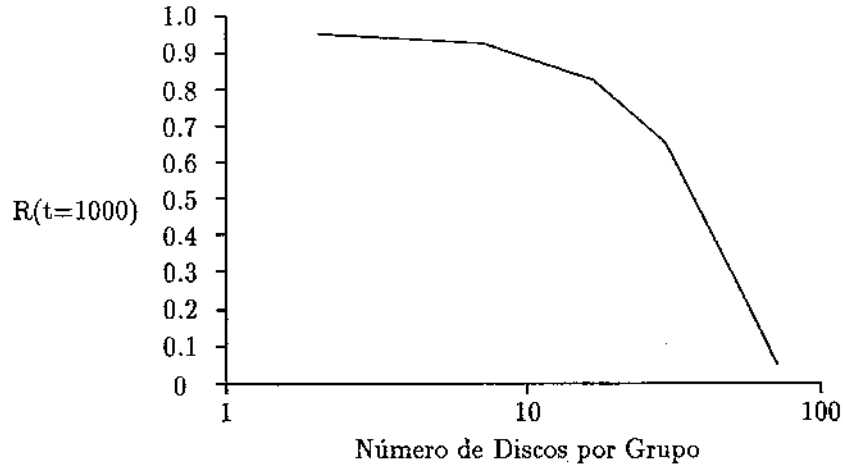

Figura 4.27: Confiabilidade Versus Número de Discos Por Grupo ($N = 8$)

se conseguir a confiabilidade desejada. Se o MTTF de um único disco for aumentada para além de um certo valor, o ganho na confiabilidade da matriz não é significativo. Assim, discos altamentes confiáveis não são a solução para matrizes de discos altamente confiáveis. A redução do tempo de recuperação de erros não produz ganhos significativos na confiabilidade da matriz. Contudo, um aumento tremendo na confiabilidade da matriz pode ser conseguido com a melhoria da cobertura de falhas. O aumento nas dimensões da matriz resulta em degradação da confiabilidade. Desse modo, a redundância de *hardware* deve ser aumentada para manter a confiabilidade da matriz.

4.5 Desempenho da Arquitetura RAID 5

Nesta seção, apresentamos uma análise sucinta do desempenho da arquitetura RAID 5 [Bit 88]. Ela supõe que as rotações dos discos constituintes da matriz sejam sincronizadas, que os processadores, canais de comunicação e controladoras sejam infinitamente rápidos, e que exista um acesso uniforme às informações no disco. No início de cada requisição, os braços estão randomicamente posicionados. A análise do caso geral é bastante complicada, e não está completa ainda.

O *seek time* é uma função da distância percorrida em cilindros. Para os discos modernos, o braço acelera e desacelera sobre 20% dos cilindros, de modo que o *seek time* pode ser modelado por uma raiz quadrada. Além dos 20%, o *seek time* é modelado por uma função linear. Por exemplo, *seek time* do disco Tandem XL80 pode ser aproximado por:

$$seek\ time\ (\tfrac{distance}{cylinders}) = \begin{cases} 5 + 0.64\sqrt{distance} & \text{se } distance < cutoff \\ 14 + \frac{distance - cutoff}{50} & \text{de outra forma,} \end{cases}$$

com uma precisão de 5%, onde *cutoff* vale 20% dos cilindros do disco.

É conveniente ter uma fórmula para a distância esperada de A braços a um cilindro particular. Supondo que, inicialmente, cada braço esteja randomicamente posicionado, Bitton [Bit 88] derivou que:

$$seek(A) = cylinders\ (1 - \frac{2}{3} * \frac{4}{5} * \frac{6}{7} * \cdots * \frac{2A}{2A+1})$$

Para a matriz RAID 5 com N discos, uma leitura de S unidades de faixa envolve A discos, onde

$$S = \frac{request\ size}{block\ size} \text{ e } A = min(S, N-1).$$

Todos os discos devem se posicionar de modo a que a transferência possa começar. O *seek time* da matriz é o máximo *seek time* individual. Uma vez que o *seek* esteja completo, a leitura espera uma rotação média e, então, transfere a uma taxa de $A * transfer\_rate$. Assim, a resposta (em segundos) de um disco RAID 5 a leituras é dada por:

$$Tempo\ de\ Leitura = seek(A) + \frac{rotate}{2} + \frac{request\_size}{A * transfer\_rate}$$

Como A discos estão envolvidos na transferência, o custo da leitura (em disco segundos) é A vezes o tempo de leitura:

$$Custo\ da\ Leitura = A * Tempo\ de\ Leitura.$$

Para escritas, um disco extra é envolvido se $A < N - 1$, de modo que definimos $A' = \min(S - 1, N - 1)$. Todos os A' discos devem se posicionar, girar até a paridade. A paridade deve ser lida e a nova paridade computada durante a rotação. Então uma escrita de dados pode ser feita a uma alta taxa de transferência. As equações são:

$$Tempo\ de\ Escrita = seek(A') + 1.5 * rotate * \frac{request\_size}{A * transfer\_rate},$$

$$Custo\ da\ Escrita = A' * Tempo\ de\ Escrita.$$

4.6 Exemplos de Implementações Comerciais

Atualmente, existem diversas implementações comerciais de matrizes RAID 5[9]. Esta tecnologia rapidamente saiu do ambiente acadêmico, e vem proliferando no mercado de estações de trabalho e de redes locais para PCs. Esta seção apresenta dois produtos, bem aceitos no mercado.

4.6.1 Raider

Raider, da Dynatek, apresenta o melhor desempenho global quando se leva em consideração flexibilidade e economia. Ele provê capacidade de troca *on-the-fly* dos discos da matriz. Consiste de uma torre compacta que pode acomodar até cinco combinações de discos de 5 1/4" com fontes de alimentação. Como cada disco possui sua própria fonte de alimentação, não existe uma fonte de alimentação principal que pudesse ser um ponto único de falha do sistema.

Se for necessário começar com uma matriz pequena, podemos configurar Raider com um único disco e adicionar módulos posteriormente. A desvantagem é a necessidade de reformatação dos discos da matriz.

Raider mostra uma mensagem na tela do servidor quando um disco falha, e cria uma entrada em um arquivo de *log*. Um utilitário restaura o conteúdo do disco com falha.

O preço médio de Raider é de US$ 33000, e pode ser usado com Netware 3.1x, OS/2 e AIX3.2.

4.6.2 Raidion

Raidion, da Micropolis, é uma matriz RAID 5 onde cada disco está em um módulo separado que contém uma fonte de alimentação e uma controladora. Os módulos, disponíveis em tamanhos que variam de 340MB a 1.7GB, podem ser unidos para formar uma torre com, no máximo, quatro módulos.

O preço de Raidion é relativamente baixo quando comparado a Raider, US$ 21000. A desvantagem é que Radion trabalha apenas com Netware 3.1x.

Radion provê indicação na tela do servidor de que algum disco apresenta defeito. Esta indicação não é escrita no *log* de erros do servidor, significando que esta informação pode ser perdida antes que se tenha a oportunidade de vê-la. Existe a possibilidade de se especificar a taxa de reconstrução dos dados antes da restauração ser iniciada. Durante todo o processo de restauração, o progresso do processo de reconstrução dos dados é exibido na tela.

[9]Quando iniciamos este trabalho de tese, há dois anos atrás, as diversas arquiteturas RAID ainda pertenciam ao ambiente acadêmico.

Capítulo 5

Um Simulador para a Arquitetura RAID 5

Neste capítulo, apresentamos um simulador da arquitetura RAID 5 para as versões 3.1 a 6.0 do sistema operacional MSDOS, implementado como *device driver* de bloco. Apresentamos também um histórico das decisões de projeto e um histórico do desenvolvimento da implementação.

As diversas técnicas sob MSDOS, aqui apresentadas, como por exemplo reentrar o MSDOS a fim de fazer uso de seus recursos, são em si um subproduto importante desta tese. Muitas aplicações não triviais do tipo *device driver*, como por exemplo o STACKER, podem se beneficiar delas em sua codificação. As técnicas apresentadas aparentemente são novas, no sentido em que não conhecemos qualquer sistema comercial que as tenha empregado, nem tampouco qualquer revista técnica sobre MSDOS que as tenha publicado. Há inclusive, uma referência recente [Ton 90] que afirma não ser possível a construção de um *device driver* que reentre o MSDOS.

5.1 Introdução

Após pesquisar sobre diversos assuntos a fim de definir um tema para esta dissertação de mestrado, chegamos à arquitetura RAID 5.

Pensamos em detalhar esta arquitetura e construir um protótipo físico de um subsistema deste tipo. Contudo, abandonamos esta idéia devido a questões econômicas e devido à complexidade do processo de importação dos principais componentes eletrônicos. Resolvemos, então, construir um simulador para esta arquitetura.

O objetivo principal que guiou a especificação do simulador foi o desejo de que a maior parte do seu código fosse utilizável, sem nenhum tipo de alteração, num protótipo físico que porventura viesse a ser construído.

Logo percebemos que um requisito importante para o simulador RAID 5 seria que o disco lógico implementado fosse indistinguível, do ponto de vista dos aplicativos e dos usuários do sistema, de qualquer disco físico existente no sistema. Isto para evitar que programas de teste especiais tivessem que ser desenvolvidos. Seria extremamente agradável testar o funcionamento do simulador com comandos do *shell* do sistema operacional empregado. Segundo esta ótica, a única solução possível seria implementar o simulador como sendo um *device driver*.

Restava então definir o sistema operacional para o qual o simulador seria desenvolvido. Inicial-

mente, pensou-se no UNIX. Como a incorporação de um novo *device driver* ao UNIX requer que seu código seja adicionado ao núcleo, precisaríamos de uma máquina com disco local para desenvolvimento. Esta máquina não poderia estar integrada à rede do nosso departamento (DCC), composta de diversas estações de trabalho SUN-*Sparcstation*, a fim de não prejudicar o trabalho dos demais usuários da rede[1]. Como o DCC possui um número reduzido de máquinas com disco local, e todas são servidoras de arquivos, esta alternativa não nos pareceu razoável. Poderíamos ter partido para um UNIX para a plataforma INTEL. Não o fizemos pois o processo de adição de um novo *device driver* ao UNIX é bastante demorado. Qualquer problema que levasse a uma queda do sistema, necessitaria que o núcleo original fosse restaurado, que o *device driver* fosse corrigido, recompilado e religado ao núcleo. Tudo isso levaria mais que dez minutos. Muito tempo, considerando que essa operação tinha probabilidade elevadíssima de ocorrer incontáveis vezes.

Descartado o UNIX, o MSDOS se tornou o candidato natural. Sabíamos que muitos problemas surgiriam dessa escolha. A estrutura interna do MSDOS muda muito de uma versão para a outra, fazendo com que código especial devesse ser escrito para cada versão. Além disso, o MSDOS não é multi-tarefa nem reentrante. Contudo, o desenvolvimento do simulador seria menos penoso, já que o MSDOS permite carregar qualquer *device driver* no instante de sua inicialização (*boot time*).

Resolvemos então adotar o MSDOS como o sistema operacional sob o qual nosso simulador seria desenvolvido. Como o MSDOS não é multi-tarefa, esta decisão determinou que não pudéssemos simular o paralelismo de leitura e de escrita da matriz RAID 5. Contudo, o código do simulador foi escrito de modo a ter essa característica se este for transportado para um sistema operacional multi-tarefa, ou para um núcleo multi-tarefa.

O objetivo deste simulador nunca foi o de medir performance da arquitetura RAID 5. Essas medições já foram extensivamente feitas ([Che 89], [Lee90], etc.). Desse modo, o fato de não se ter um paralelismo real no acesso à matriz não vem de encontro às metas desejadas.

5.2 A Arquitetura do Simulador

A premissa principal do simulador, desejo de que a maioria do seu código pudesse ser empregado, sem alteração de qualquer espécie, em um *hardware* de controle dedicado, levou-nos à conclusão de que o simulador deveria ser composto por três módulos independentes, como mostrado na Figura 5.1. O primeiro, implementaria todo o código intrínseco a um *device driver* de bloco. Faria a interface entre o núcleo do MSDOS e o segundo módulo, o núcleo do simulador. O segundo módulo não saberia que estaria sendo executado num contexto de *device driver*, nem que tipo de dispositivo de armazenamento secundário estaria de fato comandando. O terceiro módulo seria o responsável por executar o mapeamento dos discos lógicos, vistos pelo núcleo do simulador, em entidades físicas. O mecanismo de comunicação entre os diversos módulos do simulador seria a troca de mensagens.

Baseados na Figura 5.1, precisávamos decidir qual seria a entidade física, alvo do mapeamento executado pelo terceiro módulo do simulador. Para que ficássemos livres de detalhes de *hardware*, decidimos que as entidades alvo seriam arquivos no sistema de arquivos do MSDOS.

Como mapeamos os discos lógicos da matriz RAID 5 em arquivos MSDOS, nada mais natural que pedir ao próprio MSDOS que execute as operações de escrita e leitura nesses arquivos. Essa idéia, embora aparentemente simples, é de difícil implementação pois, se assim for feito, o MSDOS

[1] Como estaríamos operando a nível do núcleo do UNIX, seria muito fácil provocar uma pane total nessa máquina, a ponto dela precisar ser reinicializada.

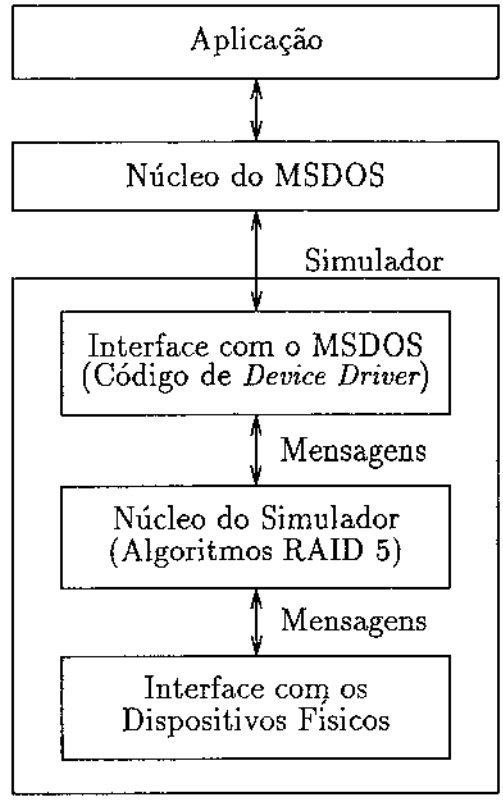

Figura 5.1: Arquitetura do Simulador RAID 5 - Primeira Idéia

deverá ser reentrado, como pode ser visto na Figura 5.2. Como sabemos, todas as referências sobre MSDOS afirmam que ele não é reentrante e que temos que conviver com este fato.

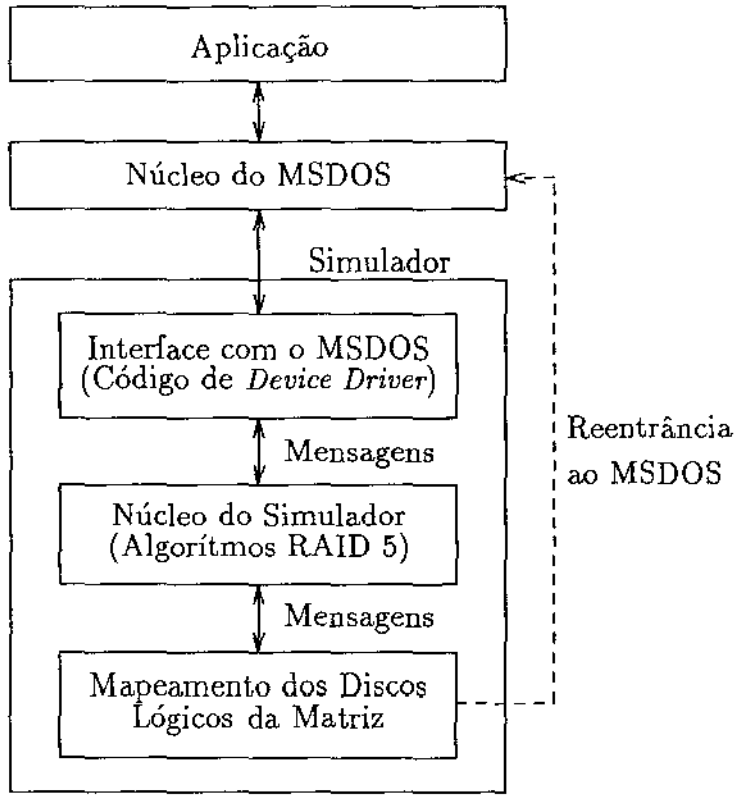

Figura 5.2: Arquitetura do Simulador RAID 5 - Versão Final

A solução mais óbvia para este problema seria escrever funções customizadas para acesso ao sistema de arquivos do MSDOS. Contudo, ela não seria uma solução adequada pois teríamos que escrever muito código colateral à implementação do simulador e, principalmente, correríamos o risco de destruir seguidamente o sistema de arquivos local devido a erros na implementação. Decidimos, então, implementar a reentrância ao MSDOS.

O motivo do MSDOS não ser reentrante é o fato dele usar posições fixas de memória para armazenar a maioria de suas informações privadas. Pensamos, então, em fazer o seguinte: quando desejássemos reentrar o MSDOS, salvaríamos estas informações em um lugar seguro, o reentraríamos e, após o retorno, restauraríamos os conteúdos originais. A implementação desta idéia (aparentemente simples) é muito difícil e intrincada.

Como o simulador manipula com arquivos, concluímos que ele necessitaria de um *program segment prefix* (PSP). A razão disso é que é nesta área que fica armazenada a *job file table* (JFT), que associa arquivos abertos no sistema de arquivos do MSDOS a *handles* de arquivos. Os *handles* na verdade são índices na JFT. Na posição `JFT[Handle]`, encontra-se um índice para uma tabela global de arquivos mantida pelo MSDOS e conhecida por *system file table* (SFT). O tamanho

máximo dessa última tabela é determinado pelo parâmetro FILES no config.sys.

Contudo, sabemos que ao instalar um *device driver*, especificado no config.sys através do parâmetro DEVICE, o MSDOS não constrói um PSP para o código do *device driver*. Outro problema de difícil solução.

A última grande dificuldade surgiu da escolha da linguagem de implementação. Resolvemos que deveria ser a linguagem C[2], pois seria extremamente difícil desenvolver algo tão complexo em *assembly*. O problema é que para as funções da biblioteca *run-time* do C operarem convenientemente, o código de inicialização (*start-up code*) do C deve ser executado. Porém, a carga convencional de um *device driver* não executa código algum desse *device driver*. Além do mais, não podemos chamá-lo após a instalação do *device driver* pois esse altera coisas importantes tais como pilha, *heap* local, etc., que não podem ser alteradas após a instalação.

Decidimos, então, que o simulador (*device driver*) deveria se instalar a partir da linha de comando do MSDOS, e não através do config.sys. Além de podermos instalar o simulador a qualquer momento, nos asseguraríamos que o código de inicialização das bibliotecas *run-time* do C seria executado e, além disso, teríamos à disposição um PSP. Outra grande vantagem decorrente dessa decisão foi o fato de não nos preocuparmos com o modelo de memória (TINY, SMALL, COMPACT, LARGE ou HUGE) empregado no desenvolvimento do simulador. O carregador do MSDOS faz a carga do simulador independentemente do modelo de memória no qual foi escrito. Se usássemos o mecanismo de carga tradicional (config.sys) teríamos sérios problemas se o modelo de memória não fosse TINY.

Esta idéia, aparentemente simples, representou um grande desafio de implementação. Tivemos que imitar corretamente o comportamento do command.com ao processar um comando DEVICE, presente no config.sys. Isto para que nenhuma aplicação percebesse qualquer inconsistência interna no MSDOS.

Após pesquisar bastante em revistas especializadas, livros, BBSs, ftp *sites* e espionar bastante o funcionamento do command.com, reunimos as informações que tornaram viáveis a implementação das duas principais características de nosso simulador: reentrar o MSDOS e se instalar a partir da linha de comando.

A fim de tornar claro o detalhamento da operação interna do primeiro módulo do simulador, apresentamos, no Apêndice A, uma visão geral da estrutura e da operação de um *device driver* de bloco, bem como uma visão geral das estruturas internas do MSDOS, utilizadas no simulador. **Recomendamos a leitura do Apêndice A, a fim de facilitar o entendimento das subseções seguintes.**

5.2.1 O Módulo de Interface com o MSDOS

Como dissemos anteriormente, o simulador RAID 5 foi implementado como sendo um *device driver* de bloco para o sistema operacional MSDOS. Este módulo é o responsável pela adição do simulador à cadeia de *device drivers* do MSDOS e o responsável pela emulação do protocolo próprio deste tipo de componente do sistema. Além disso, incorpora o código necessário para permitir a reentrância ao MSDOS.

[2]Microsoft C/C++ 7.0

Incorporação do Simulador à Cadeia de Device Drivers do MSDOS

Após o MSDOS ser inicializado, a cadeia de *device drivers* está formada. Como existe uma relação muito estreita entre todas as estruturas de dados apresentadas no Apêndice A, uma alteração nesta cadeia geralmente implica em alterações em várias dessas estruturas. Caso qualquer dessas alterações seja defeituosa, o MSDOS certamente entrará em colapso imediato, gerando a necessidade de reinicialização da máquina.

O seguinte pseudo-código mostra, de uma forma bem geral, como o *device driver* que implementa o simulador RAID 5 se instala no sistema, após ser carregado pelo MSDOS:

```
main ()
{
    if (Version < 3.1 || Version > 6.0)
        Error (WRONG_DOS_VERSION);
    GetDosVariables ();
    if (NewDrive > LastDrive)
        Error (TOO_MANY_DRIVES);
    if (InitDOSSwap () == FALSE)
        Error (DOS_SWAP_ERROR);
    if (InitDriver () == TRUE) {
        PutBlockDevice ();
        FixDOSChain ();
        GoTSR ();
    } else Error (INIT_FAILURE);
}
```

Inicialmente, testamos se a versão do MSDOS está na faixa permitida. Se não estiver, interrompemos a carga do simulador. Se estiver, obtemos algumas variáveis importantes do núcleo do MSDOS. Em seguida, duas delas (`NewDrive` e `LastDrive`) são comparadas a fim de determinarmos se as estruturas internas do MSDOS comportam a adição que desejamos fazer. Se não comportarem, interrompemos a carga do simulador. Se comportarem, tentamos inicializar as estruturas de dados necessárias à reentrância ao MSDOS. Se esta inicialização falhar, interrompemos a carga do simulador. Se for bem sucedida, chamamos o código de inicialização do *device driver* (simulador). Se este código falhar, interrompemos a carga do simulador. Se for bem sucedido, alteramos várias estruturas de dados do MSDOS, anexamos o simulador à cadeia de *device drivers* e tornamos residente o código do simulador. Atingido este ponto, o simulador está instalado e já pode ser usado como qualquer outro disco do sistema. Inclusive, *softwares* de diagnóstico tais como PCTools e Norton Utilities vêem o simulador como se fosse um disco físico real, indistinguível dos demais.

A explicação do parágrafo anterior é bastante genérica. Vamos nos aprofundar mais nela, detalhando os diversos passos da instalação do simulador.

O primeiro ponto a ser considerado é o porquê de só permitir a carga do simulador se a versão do MSDOS estiver entre 3.1 e 6.0. A resposta é muito simples. Não podemos utilizar versões anteriores à 3.1 pois várias funções do MSDOS que foram empregadas no código do programa simplesmente não existem para essas versões. Em especial, a SDA (*swappable data area*) não existe, inviabilizando o processo de reentrância[3]. Não permitimos versões superiores à 6.0 pois não podemos garantir que,

[3]Pelo menos segundo nossa abordagem.

quando do seu lançamento, a Microsoft manterá inalteradas todas as estruturas de dados usadas pelo simulador.

As variáveis do núcleo do MSDOS que são utilizadas pelo simulador estão listadas a seguir:

- `NulDrv`: apontador para a cabeça da cadeia de *device drivers*; É usado por `FixDOSChain ()` para inserir o *device header* (cabeçalho) do simulador na cadeia, logo após o dispositivo `NUL`; Esta é a posição empregada pelo MSDOS quando este acrescenta novos *device drivers* via `config.sys`; `NUL` deve ser sempre o primeiro da cadeia, pois, se assim não o for, o MSDOS fica inconsistente e sua operação imprevisível;

- `NewDrive`: índice da primeira entrada não usada na matriz de CDSs (*current directory structure*); Isto reflete, para o usuário do simulador, na letra que estará associada ao disco RAID 5; Por exemplo, se o índice for quatro, então o usuário verá o disco RAID 5 como sendo o disco `D:`;

- `LastDrive`: índice associado à última entrada válida na matriz de CDSs;

- `CDSArray`: apontador para a matriz de CDSs; É utilizado por `PutBlockDevice ()` para preencher os campos da entrada `CDSArray[NewDrive]`, se possível;

- `CDSSize`: tamanho de cada entrada na matriz de CDSs; Este número varia de acordo com as versões do MSDOS suportadas por nossa implementação.

Todas essas variáveis são obtidas a partir da LOL (*list of lists*), via chamada a `INT 21h` (*dispatcher* do MSDOS), subfunção `0x52`. A localização dessas variáveis depende da versão corrente do MSDOS, ou seja, não é estável entre as versões 3.1 a 6.0.

Existe a necessidade do teste

```
if (NewDrive > LastDrive)
    Error (TOO_MANY_DRIVES);
```

pois, neste caso, não existe entrada disponível na matriz de CDS para se acrescentar outro *device driver* de bloco. Na verdade, esta é uma alternativa um pouco radical, pois é possível se reconfigurar o MSDOS *on-the-fly* para que ele comporte mais *device drivers* de bloco. Por exemplo, poderíamos requisitar ao MSDOS uma área de memória maior que aquela ocupada pela matriz de CDSs original, fazer uma cópia desta matriz na área recém alocada, alterar a LOL para apontar para ela, etc., etc. e etc. Embora saibamos como fazer isto, não o fizemos pois isto fugiria ao espírito deste trabalho. Isto pode ser implementado em versões futuras do simulador.

Discutiremos `InitDOSSwap ()` mais para frente, quando abordarmos o processo de reentrância ao MSDOS.

O procedimento `InitDriver ()` cria uma pilha local para o simulador, acha o primeiro segmento livre da memória após o código do simulador, verifica se o disco já existe, o cria se não existir, e exibe uma mensagem com a identificação do simulador e a letra pela qual o disco RAID 5 será referenciado. A mensagem aparece, por exemplo, da seguinte forma:

```
c:\> raid5                --> invoca o simulador

Simulador RAID 5 -- Versao 1.0
```

```
(C) Copyright Hermano 1991-1993
Instalado como Disco E:

c:\>
```

Há uma necessidade de pilha local pois a pilha do MSDOS é muito pequena. Como é o próprio MSDOS que chama o código do simulador, o fato de poder haver reentradas faz com que a possibilidade de estouro de pilha seja muito grande. Se isto ocorrer, o sistema entra imediatamente em colapso. O primeiro segmento de memória livre é usado por `GoTSR ()` para preservar a área utilizada pelo simulador de usos futuros pelo gerenciador de memória do MSDOS.

O procedimento `PutBlockDevice ()` é o responsável pela atualização das estruturas de dados do núcleo do MSDOS que são associadas a *device drivers* de bloco. Inicialmente, criamos o DPB (*disk parameter block*) do dispositivo a partir de seu BPB (*BIOS parameter block*), via `INT 21h`, subfunção `0x53`. Preenchemos alguns de seus campos, tais como o índice na matriz de CDSs associado a esse DPB, e o acrescentamos ao final da lista de DPBs. Em seguida, preenchemos a entrada correspondente na matriz de CDSs. Finalizando, atualizamos a LOL para que esta reflita a presença de mais um *device driver* de bloco no sistema. Incrementamos o contador de tais *device drivers*.

O procedimento `FixDOSChain ()` é bastante simples. Utiliza um algoritmo de inserção em lista simplesmente ligada. Este é o ponto final do processo de atualização das estruturas de dados do MSDOS para que este suporte mais um *device driver* de bloco.

Finalmente, o procedimento `GoTSR ()` é usado para encerrar o processo de carga do simulador. Devolve a área alocada para o *environment* do simulador ao MSDOS, o excedente da área alocada pelo MSDOS para o simulador e conclui o processo ficando residente. Não há necessidade de *environment* pois não fazemos uso de nenhuma variável de ambiente. Devolvemos memória ao MSDOS pois ele sempre aloca mais espaço para um programa do que o necessário. Ele faz isto para que o programa possa fazer alocações dinâmicas, executar outros programas, etc. Como não fazemos nada disto, calculamos a memória estritamente necessária, via informação passada por `InitDriver ()`, e devolvemos o excedente.

A Figura 5.3, mostra as estruturas de dados do MSDOS que são alteradas no processo de adição *on-the-fly* de um *device driver* de bloco ao sistema.

Processo de Reentrância ao MSDOS

É muito comum, em publicações especializadas sobre MSDOS, ver a seguinte afirmação: o MSDOS não é reentrante para as funções acima da subfunção `0x0C`, com raras exceções. A razão disso é simples. O MSDOS possui três pilhas internas. Uma para operação normal, outra para operações de I/O e outra para processamento de erros críticos. Se nenhum evento fizer o MSDOS mudar para outra pilha, dentre as três citas, ele permanece usando uma mesma pilha. Além disso, sempre que seu serviço é solicitado, ele começa a usar as pilhas como se não houvesse nada nelas. Assim, quando uma função reentra o MSDOS, o contexto da chamada anterior é destruído. Isto causa colapso no sistema. Além disso, o MSDOS utiliza posições fixas de memória para armazenar suas variáveis.

A BIOS do computador também não é reentrante. Neste caso, a razão é que ela usa posições fixas de memória para armazenar suas variáveis.

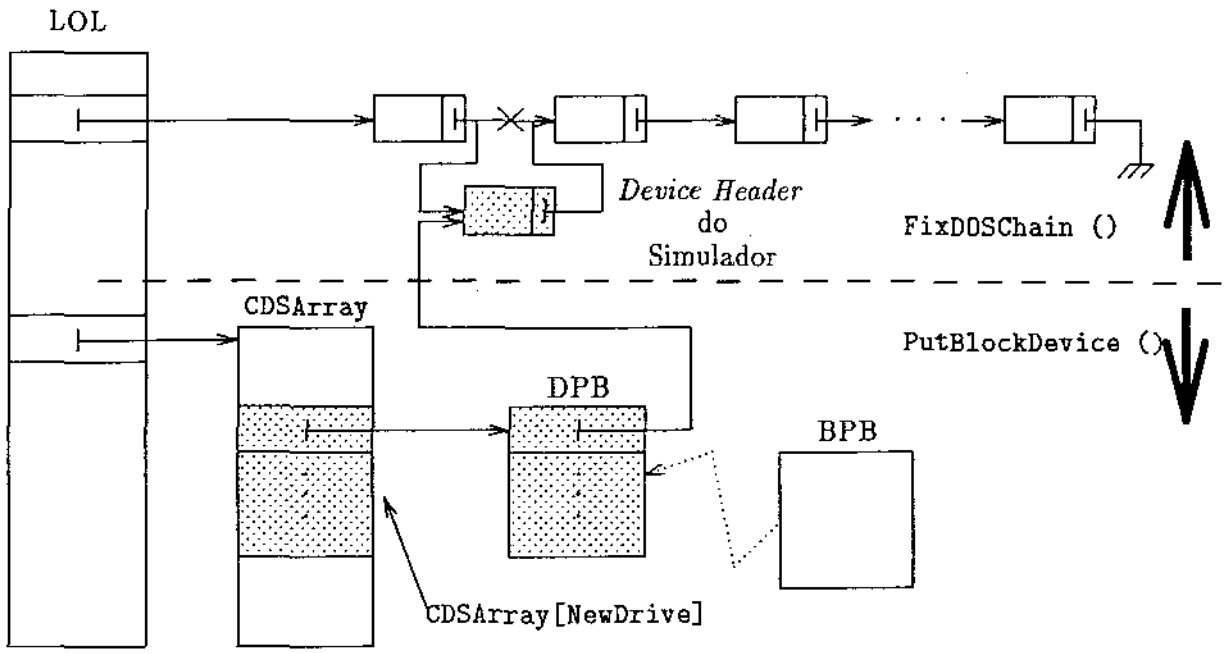

Figura 5.3: Adição *on-the-fly* de um *Device Driver* de Bloco ao MSDOS

A solução encontrada para esses dois problemas, que impedem a reentrância ao MSDOS, foi salvar essas áreas do processo interrompido e restaurá-las após a conclusão do novo processo. Isto pode ser feito de forma muito eficiente, como apresentado a seguir.

Codificamos três funções para serem usadas pelo código do simulador, com a finalidade de auxiliar no processo de reentrância ao MSDOS. São elas:

- `InitDOSSwap ()`: inicializa as estruturas de dados necessárias à troca das áreas de dados do MSDOS e da BIOS;
- `SaveDOSSwap ()`: salva as áreas de dados em uso corrente;
- `RestoreDOSSwap ()`: restaura as áreas de dados salvas por `SaveDOSSwap ()`.

Essas três funções são dependentes de versão do MSDOS. Por exemplo, nas versões 3.1 a 3.3, 5.0 e 6.0, a SDA pode ser obtida via `INT 21h`, subfunção `0x5D06`. Já, na versão 4.x, podemos obtê-la via `INT 21h`, subfunção `0x5D0B`. Além disso, possuem estruturas diferentes.

A função `InitDOSSwap ()` aloca espaço de memória suficiente para manter uma cópia da SDA e uma cópia da área de dados da BIOS, adequada para a versão corrente do MSDOS. Se a alocação falhar, `InitDOSSwap ()` retorna falso. Caso contrário retorna verdadeiro. Lembremos que se essa função falhar, o simulador não pode ser instalado. O programa principal do simulador cuida desse fato.

A função `SaveDOSSwap ()` tem a responsabilidade de salvar tanto a área de dados do MSDOS quanto a da BIOS. Essa é uma tarefa essencialmente trivial, já que sabemos como localizar e qual

é a extensão dessas duas áreas. Só temos que nos preocupar com o fato de que existe uma situação em que não podemos manipular com a SDA. O momento em que o MSDOS está em uma seção crítica. Como este fato é extremamente raro, nosso simulador praticamente sempre pode reentrar o MSDOS. A entrada e a saída de uma seção crítica é sinalizada pelo MSDOS via chamadas a `INT 2Ah`, com subfunções apropriadas.

A função `RestoreDOSSwap ()` é simples. Ela restaura as áreas de dados que foram salvas. O processo interrompido, quando retomado, não fica sabendo do que aconteceu. Tanto seu estado interno quanto o do MSDOS estão consistentes e são os mesmos que antes da troca de contexto.

Emulação do Protocolo de Device Driver de Bloco

O Apêndice A apresenta a estrutura de um *device driver* de bloco, o protocolo de comunicação com o MSDOS (implementado pela rotina de estratégia e pela rotina de interrupção) e as funções por ele suportadas (`Read`, `Write`, etc).

A função da rotina de estratégia é armazenar, numa variável local ao simulador, um apontador para o *request header* criado pelo MSDOS para a execução do serviço que se está requisitando. Sua implementação é trivial:

```
void __far __loadds Strategy (void)
{
    __asm {
        MOV     WORD PTR RequestHeader, BX
        MOV     WORD PTR RequestHeader + 2, ES
    }
}
```

Esta rotina tem o atributo `__far` pois o MSDOS espera que ela retorne via uma instrução de retorno tipo FAR (32 *bits*). O atributo `__loadds` serve para indicar ao compilador que carregue o registrador de segmento de dados (`DS`) com o valor do segmento de dados do simulador. Se isso não fosse feito, o funcionamento do simulador seria imprevisível pois, praticamente sempre, a variável `RequestHeader` estaria apontando para um lugar diferente daquele indicado pelo MSDOS. Isto porque o MSDOS não ajusta o valor de `DS` antes de chamar o código do *device driver*.

O código da rotina de interrupção é consideravelmente mais complicado. A seguir, apresentamos seu pseudo-código:

```
void __far __loadds __saveregs Interrupt (void)
{
    SaveFlags ();
    ChangeToLocalStack ();
    _fmemmove (&ReqHdr, RequestHeader,
               sizeof (BLOCK_REQUEST_HEADER));
    if (!SaveDOSSwap ()) {
        ReqHdr.ReturnedStatus = ERROR | GENERAL_FAILURE | DONE;
        goto EXIT;
    }
    SetPSP (_psp);
    SetDTA (_psp);
```

```
        ReturnedStatus = (ReqHdr.CommandCode > MAX_CMD) ?
                         ERROR | UNKNOWN_CMD | DONE :
                         (*Dispatch[ReqHdr.CommandCode]) () | DONE;
    EXIT:
        RestoreDOSSwap ();
        _fmemmove (RequestHeader, &ReqHdr,
                   sizeof (BLOCK_REQUEST_HEADER));
        ChangeToOriginalStack ();
        RestoreFlags ();
    }
```

O atributo _saveregs diz ao compilador que a primeira ação a ser tomada na execução deste procedimento é salvar todos os registradores na pilha, e a última é restaurá-los a partir da pilha. A única exceção fica por conta do registrador de `flags`, o qual tem que ser salvo explicitamente. Em seguida, mudamos para uma pilha local. A razão disto é que não queremos correr o risco de invadir áreas de código do MSDOS. Se a quantidade de pilha utilizada pelo simulador exceder o tamanho da SDA, podemos ter sobreposição de código. Então, copiamos o *request header* passado pelo MSDOS para uma variável privada do simulador[4]. Em seguida, tentamos salvar as áreas de dados do sistema, a fim de que possamos reentrar o MSDOS com segurança. Se isto não for possível, retornamos ao MSDOS com o *status* `GENERAL_FAILURE`. Isto raramente deve acontecer. De fato, em nossos testes do simulador, nunca aconteceu. Se tudo correr bem, ajustamos o PSP (*program segment prefix*) e a DTA (*disk transfer area*) corrente para que apontem para as respectivas áreas do simulador. Isto para que possamos utilizar, com segurança, as funções do MSDOS que manipulam arquivos. Em seguida, testamos a validade do comando passado pelo MSDOS. Se for válido, chamamos a função adequada ao tratamento da requisição, via uma tabela de apontadores para funções, e retornamos ao MSDOS com o *status* da função executada. Se o código não for válido, retornamos ao MSDOS com o *status* `UNKNOWN_COMMAND`. O procedimento de retorno ao MSDOS é simples: restauramos as áreas de dados do MSDOS, copiamos o conteúdo da cópia privada do *request header* no *request header* passado pelo MSDOS, retornamos à pilha original e restauramos o registrador de `flags`. Os demais registradores são automaticamente restaurados pelo compilador. Não há a necessidade de restaurar o PSP e a DTA anterior pois estas estruturas fazem parte da SDA que foi restaurada.

Existe a necessidade de se fazer uma cópia do *request header* passado pelo MSDOS para uma variável privada ao simulador, pois esta estrutura de dados reside na SDA que vai ser trocada. Se não fizéssemos isto, ao restaurarmos a SDA, perderíamos as alterações, feitas pelo simulador, no *request header* que seria devolvido ao MSDOS com o resultado da requisição. A necessidade de troca da SDA é reflexo direto da nossa abordagem para a reentrância ao MSDOS. A Figura 5.4 mostra, graficamente, o caminho percorrido pelo *request header* criado pelo MSDOS, desde quando é apresentado ao simulador até quando é devolvido ao MSDOS.

5.2.2 O Núcleo do Simulador

Este módulo do simulador é o responsável pela execução dos algoritmos RAID 5. São implementados os procedimentos de leitura e escrita de dados, bem como o procedimento de recuperação de erros. Como foi dito antes, este código independe do fato de estar imerso numa estrutura de *device driver* de bloco e se adapta bem a ambientes multitarefa.

[4] No próximo parágrafo, apresentamos o porquê deste fato.

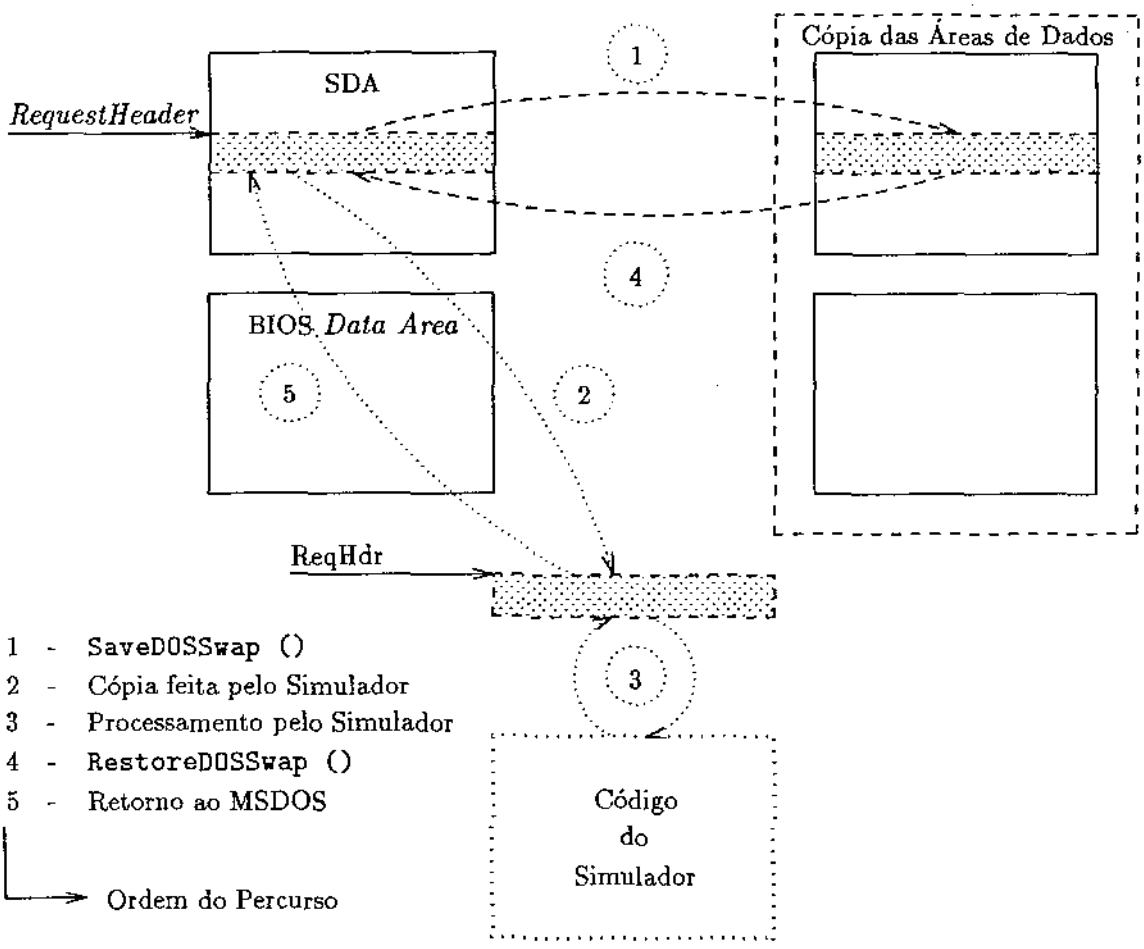

Figura 5.4: Percurso do *Request Header* no Processo de Emulação do Protocolo de *Device Driver* de Bloco

Dentre todos os posicionamentos possíveis para os dados e informação de redundância, como pode ser visto no Capítulo 4, escolhemos o assimétrico direito. Existem duas funções, `DataMapping ()` e `ParityPlacement ()`, que implementam este tipo de posicionamento. Elas recebem como parâmetro um número lógico de setor e devolvem o endereço físico correspondente. Essas funções são extensivamentes usadas em todo o código do núcleo do simulador.

A função `Read ()` é a responsável pela leitura de dados requeridos pelo MSDOS. Seu pseudo-código está mostrado a seguir:

```
static unsigned Read (void)
{
    unsigned i, j, FailCol, ErrorCount = 0;

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            if (ReadDisk (i, j) == ERROR) {
                FailCol = j;
                ErrorCount++;
            }
            if (ErrorCount > 1)
                return (ERROR | GENERAL_FAILURE);
        }
        if (ErrorCount == 1)
            if (ReconstructDisk (i, FailCol) == ERROR)
                return (ERROR | GENERAL_FAILURE);
            else {
                ReadDisk (i, FailCol);
                ErrorCount = 0;
            }
    }
    return (OK);
}
```

Suponhamos que a matriz RAID 5 seja composta de `ROWS` linhas e `COLUMNS` colunas. Para cada linha da matriz, executamos o seguinte algoritmo. Percorremos todas as colunas requisitando a cada uma delas que execute, via a função `ReadDisk ()`, a parte da leitura que lhe cabe. Se algum erro for encontrado, marcamos a coluna defeituosa e incrementamos um contador de erros. Se este contador for maior que um, é sinal que existe mais que um disco com falha nesta linha. Isto causa a falha da requisição de leitura, indicado pelo retorno com *status* `GENERAL_FAILURE`. Se houve erro em apenas um disco, pode ser possível a recuperação dos dados. Chamamos `ReconstructDisk ()` passando o número da linha e o número da coluna com falha para que se faça a tentativa de restauração. Se for bem sucedida, executamos a leitura que não pôde ser feita devido ao erro e zeramos o contador de falhas por grupo. Se for mal sucedida, retornamos com *status* `GENERAL_FAILURE`, indicando a falha da requisição. Se não houve erro, ou se houve um erro consertado, retornamos com *status* `OK`.

A função `ReadDisk ()` recebe como entrada um número de linha, um número de coluna, e os números lógicos dos setores requisitados. Verifica quais setores da requisição pertencem a este disco e os lê, se existirem. Isto nos permite tirar vantagens de um sistema multitarefa. Podemos

disparar vários processos para que efetuem as leituras simultaneamente. Se tivermos um *hardware* de controle adequado, podemos acessar os discos de modo simultâneo. Para saber se um dado setor lógico pertence a um determinado disco, basta se obter o mapeamento físico deste setor, via `DataMapping ()`, e testar se a linha e a coluna física são respectivamente iguais a linha e a coluna do disco em questão.

A função de escrita de dados é análoga a de leitura, como pode ser visto no seguinte pseudo-código:

```
static unsigned Write (void)
{
    unsigned i, j, FailCol, ErrorCount = 0;

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            if (WriteDisk (i, j) == ERROR) {
                FailCol = j;
                ErrorCount++;
            }
            if (ErrorCount > 1)
                return (ERROR | GENERAL_FAILURE);
        }
        if (ErrorCount == 1)
            if (ReconstructDisk (i, FailCol) == ERROR)
                return (ERROR | GENERAL_FAILURE);
            else {
                WriteDisk (i, FailCol);
                ErrorCount = 0;
            }
    }
    return (OK);
}
```

A diferença é que, como estamos tratando de escrita, precisamos levar em consideração o cálculo da informação de redundância. Assim, `WriteDisk ()` tem que calcular e escrever a nova paridade. Isto é feito chamando-se a função `WriteNewParity ()`. O Cálculo executado por esta última função pode ser encontrado no Capítulo 4.

O processo de recuperação de erros na arquitetura RAID 5 permite consertar erros simples e detectar erros múltiplos em um grupo. Assim, no caso de falha de mais que um disco por grupo ocorre perda irrecuperável de informação. Caso haja apenas uma falha por grupo, mesmo que em mais de um grupo, a informação ainda é recuperável. O processo de reconstrução de um determinado disco de um grupo, implementado por `ReconstructDisk ()` é simples, e pode ser encontrado no Capítulo 4.

5.2.3 Interface com Os Dispositivos Físicos

Em nossa versão do simulador, resolvemos mapear um disco lógico, visto pelos algoritmos RAID 5, em arquivos no sistema de arquivos do MSDOS. Essa decisão foi fonte de inúmeros problemas que

tiveram que ser resolvidos a fim de que essa idéia fosse viável. Os algoritmos de reentrância ao MSDOS foram desenvolvidos com esta finalidade.

Como a reentrância ao MSDOS foi conseguida, o mapeamento a que nos referimos, executado por `ReadPhysicalDisk ()` e `WritePhysicalDisk ()`, ficou extremamente fácil de ser codificado. Podemos usar as funções de manipulação de arquivos `_read ()` e `_write ()` da biblioteca *run-time* do compilador C. Por exemplo,

```
int ReadPhysicalDisk (int Row, int Col, long StartSector,
                      int SectorsToRead, void* Buff)
{
    if (_lseek (Matrix[Row][Col], StartSector * 512L,
                SEEK_SET) == -1)
        return (-1);    // erro
    return (_read (Matrix[Row][Col], Buff, SectorsToRead * 512));
}
```

As entradas da matriz `Matrix` contêm os *handles* dos arquivos associados. Por exemplo, a entrada `Matrix[1][2]` contém o *handle* do disco 2 da linha 1.

O fato de estarmos fazendo este tipo de mapeamento nos obriga a aumentar os parâmetros `FILES` e `BUFFERS` no arquivo `config.sys`. Isto deve ser feito pois, como a matriz possui muitos discos, corremos o risco de estourar as estruturas de dados internas do MSDOS e causar paralisação da máquina. Recomendamos `FILES = 100` e `BUFFERS = 50`.

Se tivéssemos escolhido, por exemplo, mapear um disco lógico em um disco físico tipo SCSI, bastaria rescrever `ReadPhysicalDisk ()` e `WritePhysicalDisk ()` de modo adequado.

5.3 Depuração do Simulador

Durante a implementação de cada fase do simulador, desenvolvemos diferentes metodologias de depuração. Além disso, após o final da codificação do simulador, fizemos também um teste de integração dos módulos.

A depuração do módulo de interface com o MSDOS foi dividida em duas partes: depuração do protocolo de *device driver* e depuração da reentrância ao MSDOS. Para a primeira parte, escrevemos um *device driver* para um RAM *disk* de $64KB$ e depuramos seu funcionamento, até que funcionou a contento. Toda a dificuldade vinha do fato de que qualquer erro na implementação do protocolo causava colapso no sistema. Isso acontecia tão logo algum serviço fosse requisitado ao RAM *disk*. Devido a isso, tinhamos muito pouco controle sobre o problema, de modo que a depuração foi baseada em tentativa e erro. A depuração da reentrância ao MSDOS também foi difícil. As razões foram as mesmas que para a depuração do protocolo de *device driver*. Neste caso, reimplementamos o RAM *disk* para que ele operasse como um FILE *disk*, isto é, um disco mapeado no sistema de arquivos do MSDOS. As operações de leitura e escrita de setores foram implementadas via requisições pertinentes ao MSDOS.

Na depuração do núcleo do simulador, criamos um programa que gerava aleatoriamente requisições de leituras e escritas. Instruções de depuração dentro deste módulo imprimiam informações relevantes[5] que nos permitiram avaliar a correção da implementação. Durante a depuração, as requisições de leituras e escritas aos discos componentes da matriz RAID 5 retornavam

[5]Tal como o mapeamento lógico para físico de cada bloco lógico envolvido na transferência.

com sucesso ou com falha de modo aleatório. Fizemos isto, para que fosse possível testar o processo de recuperação de erros. Esse módulo foi depurado isoladamente, sem estar inserido no simulador.

O mapeamento lógico para físico dos discos lógicos da matriz não precisou ser depurado. A primeira implementação já funcionou a contento. Isto se deveu à simplicidade deste código.

Como o simulador é um *device driver* de bloco, ele é indistinguível de um disco físico qualquer, desde que não tentemos acessá-lo sem a intervenção do MSDOS. Assim, o teste de integração foi feito com comandos e programas comuns do MSDOS. Por exemplo, supondo que o disco RAID 5 seja o disco `E:`, quando fizemos

```
c:> copy xx.yy e:
c:> e:
e:> comp xx.yy c:
```

obtemos uma resposta dizendo que os arquivos são iguais. Também, programas tais como o Norton Utilities e o PCTools funcionaram perfeitamente com o disco RAID 5. A única exceção foi o comando `calibrate` do Norton Utilities, que tentou acessar o disco diretamente via BIOS. Também testamos o simulador no ambiente Windows com vários gerenciadores de memória, tais como 386Max, QEMM, etc.

Como um *hard disk* é normalmente muito confiável, não pudemos testar a recuperação de erros simplesmente esperando que algum setor de um ou mais arquivos da matriz falhasse. Precisamos provocar tais falhas. A alternativa mais radical seria apagar todo um dado arquivo, indicando que existem falhas em todos os setores daquele disco. Porém, não podíamos simplesmente apagar um tal arquivo a partir da linha de comando do MSDOS, pois nunca se pode remover um arquivo que está aberto por algum programa. A solução encontrada foi escrever um programa que mandasse um comando `IOCTLWrite` para o simulador, requisitando que ele fechasse e removesse do diretório um determinado arquivo da matriz. Quando apagávamos dois discos de uma mesma linha, qualquer acesso ao disco RAID 5 falhava (`GENERAL_FAILURE`). Quando apagávamos um disco de cada linha, víamos que a matriz se recuperava.

5.4 Aplicações das Técnicas Desenvolvidas Para o MSDOS

As técnicas desenvolvidas para o MSDOS apresentadas neste capítulo podem ser empregadas na implementação de diversos aplicativos relevantes, que seriam muito difíceis de codificar utilizando os métodos tradicionais. Tais aplicativos podem ser divididos em duas categorias: *device drivers* e TSRs. Um TSR (*terminate and stay resident*) é um programa que se instala na memória do computador, sendo ativado por alguma interrupção[6], podendo funcionar concorrentemente com outros programas. Um exemplo bem conhecido de TSR é o programa Sidekick.

Na categoria de *device drivers* podemos citar três exemplos onde o uso das técnicas deste capítulo facilitariam tremendamente a codificação dos mesmos:

1. auditoria (*journaling*) do sistema de arquivos do MSDOS – Poderíamos escrever um *device driver* que anotasse em um arquivo as operações de leituras e escritas feitas sobre um ou mais arquivos, previamente especificados; Se esta facilidade for usada juntamente com um sistema de senhas (por *hardware* ou por *software*), poderíamos saber qual usuário fez o que com os

[6] Normalmente a interrupção de relógio.

arquivos marcados para auditoria; Este tipo de procedimento é muito utilizado em grandes instalações;

2. *backup* do sistema de arquivos do MSDOS – Poderíamos escrever um *device driver* que fizesse um *log* em uma fita ou em um disco dedicado para isso, de todas as alterações feitas em um determinado disco; Se o disco falhar, restauramos o *backup* inicial e executamos todos os *logs* sobre ele, de modo a recuperar o conteúdo perdido;

3. espelhamento (*mirroring*) – Poderíamos implementar facilmente RAID 1, sem necessidade de *hardware* adicional; Normalmente, todo PC tem uma controladora de disco rígido capaz de comandar dois desses discos; Bastaría que instalássemos um segundo disco e o *device driver* de espelhamento.

Na categoria de TSRs podemos citar dois exemplos de aplicação das técnicas deste capítulo:

1. *spooler* de impressão de alto desempenho – O problema com as implementações tradicionais desse tipo de *spooler*, principalmente quando se está usando o Windows, é que eles têm que aguardar o MSDOS não estar sendo usado para que dados possam ser recuperados do disco e enviados à impressora; No Windows, a probabilidade do MSDOS não estar sendo usado é muito baixa[7]; Isto explica a lentidão na impressão dentro do Windows, quando comparada à mesma impressão fora dele;

2. extensão multitarefa ao MSDOS – Poderiamos escrever um TSR que alternasse a execução de diversas aplicações MSDOS, segundo alguma política de escalonamento; Isso é possível pois sabemos determinar e alterar o PSP corrente, o qual é, para o MSDOS, um identificador de processo.

Já apresentamos a técnica de construção de *device drivers* reentrantes ao MSDOS. A subseção seguinte apresenta a técnica de construção de TSRs reentrantes ao MSDOS.

5.4.1 TSRs Reentrantes ao MSDOS

No rotina principal do TSR, aquela que o instala na memória, deve-se, antes de mais nada, inicializar as estruturas de dados que permitem a reentrada via `InitDOSSwap ()`. Se isto não for possível, o TSR não pode ser instalado.

```
main ()
{
    ShowCopyright ();
    if (!InitDOSSwap ()) {
        fprintf (stderr, "Falha na instalacao");
        exit (SDA_FAIL);
    }
    .   // Inicializacao dependente do
    .   //     TRS particular que se esta
    .   //     desenvolvendo
    GoTSR ();
}
```

[7] O Windows faz muito *swap* em disco.

Quando um TSR for ativado, devemos aguardar o MSDOS estar fora de uma seção crítica[8] e, então, mudar o contexto através de `SaveDOSSwap ()`. Se for necessário, por algum motivo, que o TSR manipule com o PSP do processo interrompido, devemos utilizar a função `GetPSP ()`.

Para que possamos manipular arquivos, sem efeitos colaterais, precisamos alterar o PSP e a DTA corrente para que apontem, respectivamente, para o PSP do programa TSR e para o *byte* cujo *offset* vale `0x80` a partir do início deste PSP. A razão principal deste requisito é que é aí que se encontra a JFT do processo associado. Como exemplo de efeito colateral, considere o seguinte exemplo: um programa A foi interrompido por um programa TSR B; Como o MSDOS não faz a troca automática de PSPs, se B abrir arquivos, na verdade estará usando handles da JFT de A; Suponha, então que o programa A termine. Neste caso o MSDOS descarta o PSP de A e, conseqüentemente, sua JFT; Deste ponto em diante, B não consegue mais acessar seus arquivos. Não há a necessidade de restaurar o PSP e a DTA anterior ao retornar do manipulador de interrupção, pois estes dados estão armazenados na SDA que será restaurada.

A partir desse momento, podemos chamar qualquer função do MSDOS sem nenhum tipo de risco. No instante de deixar o código TSR, chamamos `RestoreDOSSwap ()`. Não é necessário restaurar o PSP e a DTA do processo interrompido pois estes fazem parte da SDA que foi restaurada.

```
void __interrupt __far InterruptHandler ()
{
    if (!SaveDOSSwap ())    // Muda de contexto
            // O MSDOS esta em uma secao critica.
            //     Tente mais tarde. Este return sera
            //     executado rarissimas vezes.
        return;
    ForegroundPSP = GetPSP (); // Opcional
    SetPSP (_psp);
    SetDTA (_psp);
    .
    .
    .
    RestoreDOSSwap();       // Restaura contexto
}

unsigned GetPSP (void)
{
    __asm {
        MOV     AH, 0x62
        INT     0x21
        MOV     AX, BX
    }
}
```

[8] Raríssimas serão as vezes em que o MSDOS se encontrará em uma seção crítica.

```
void SetPSP (unsigned PSP)
{
    __asm {
        MOV     AH, 0x50
        MOV     BX, PSP
        INT     0x21
    }
}

void SetDTA (unsigned PSP)
{
    __asm {
        PUSH    DS
        MOV     AH, 0x1A
        MOV     DX, 0x80
        MOV     DS, PSP
        INT     0x21
        POP     DS
    }
}
```

A abordagem tradicional para codificação de programas TSR é bastante diferente desta proposta, pois nunca se reentra o MSDOS. O que se faz é esperar até que o MSDOS sinalize que ele está livre, ou seja, que nenhum processo esteja em execução, para que possamos requisitar seus serviços. Obviamente, esta técnica é bastante limitada quando comparada à proposta. Em particular, não poderíamos ter escrito o simulador usando tal técnica, pois as funções do MSDOS seriam chamadas de dentro de um *device driver*, o qual estaria ocupando a atenção do MSDOS.

Capítulo 6

Considerações Finais e Trabalhos Futuros

Neste capítulo, apresentamos as conclusões deste trabalho, seguidas de algumas sugestões visando futuros estudos nas áreas abrangidas por este projeto.

6.1 Considerações Finais

Neste trabalho foi desenvolvido um simulador para a arquitetura de subsistemas de discos magnéticos RAID 5, sob o sistema operacional MSDOS versão 3.1 a 6.0, implementado com um *device driver* de bloco.

Para que o simulador fosse desenvolvido, vários passos intermediários tiveram que ser percorridos:

- determinação da natureza do simulador: um requisito importante para o simulador é que o disco lógico implementado seja indistinguível, do ponto de vista dos aplicativos e dos usuários, de qualquer disco físico presente no sistema; Isto para evitar que programas especiais tivessem que ser desenvolvidos; A solução encontrada foi implementar o simulador como sendo do tipo *device driver*;

- determinação do sistema operacional alvo do *device driver*; Escolhemos o MSDOS por razões de facilidade de teste e desenvolvimento, bem como disponibilidade de máquina;

- estudo profundo da arquitetura interna do MSDOS em suas diversas versões; Como decidimos mapear os discos físicos da matriz RAID 5 em arquivos no sistema de arquivos do MSDOS, o simulador necessariamente reentra o MSDOS; Isto geralmente não é possível de se fazer de forma documentada pela Microsoft; Contudo, um domínio das estruturas internas das diversas versões do MSDOS nos permite alterá-las profundamente, de modo que este fato se torne possível; Além disso, o simulador é um *device driver* que se instala a partir da linha de comando do MSDOS;

- estudo dos algoritmos relativos ao posicionamento escolhido dos dados e da paridade da matriz RAID 5, bem como o correspondente algoritmo de recuperação (*on-the-fly*) de erros.

Alguns resultados deste trabalho são relevantes no contexto MSDOS e outros no contexto RAID 5. No contexto MSDOS, podemos citar:

- desenvolvimento de código para que um *device driver* se instale a partir da linha de comando do MSDOS; A importância disso é permitir que o código do simulador seja escrito facilmente em C, podendo usar suas bibliotecas *run-time* (as quais necessitam que seja executado o código de inicialização do C); A carga de um *device driver*, a partir do `config.sys`, não permite que isto aconteça;

- desenvolvimento de código que permite a reentrância ao MSDOS; Este fato tem muita relevância prática pois permite que programas muito complexos (tais como o simulador RAID 5) que requerem muitos serviços já existentes no MSDOS possam executar em *background* ou a nível de *device driver* sem nenhum problema; Ressaltamos que a abordagem clássica de construções de aplicativos TSR não permite a reentrada ao MSDOS; Existe um *flag* (chamado INDOS) que nos permite saber se existe ou não algum processo sendo servido pelo MSDOS ou não; Na abordagem clássica, temos que aguardar até que o MSDOS esteja livre para que possamos requisitar seus serviços.

No contexto da arquitetura RAID 5 podemos citar o seguinte fato como sendo relevante:

- como o núcleo do simulador é totalmente independente dos outros dois módulos, podemos utilizá-lo, sem alteração de qualquer espécie, em um *hardware* de controle dedicado; Para tanto, só necessitamos escrever o equivalente dos dois outros módulos; Temos que permitir que a controladora converse com seu *host* e que possa acessar os discos físicos da matriz alvo do mapeamento dos discos lógicos vistos pelo núcleo do simulador.

Resumindo, apresentamos um simulador que utiliza técnicas muito importantes à nível de programação MSDOS e que parte de seu código pode ser utilizado em uma implementação de *hardware*, a saber, o módulo que contém os algoritmos RAID 5.

6.2 Trabalhos Futuros

Na seção anterior, comentamos os resultados relevantes que foram obtidos no contexto MSDOS e no contexto RAID 5. Eles podem ser explorados mais ainda, a fim de se obter resultados mais profundos.

No contexto MSDOS podemos citar:

- elaboração de um *kit* de desenvolvimento de *device drivers* e programas residentes (TSR), em C, que possam reentrar o MSDOS;

- elaboração de ferramentas de análise das estruturas internas e dos parâmetros do MSDOS;

- elaboração de ferramentas que possam alterar os parâmetros do MSDOS com o sistema já inicializado (por exemplo, aumentar o número máximo de arquivos abertos simultaneamente);

- escrever uma extensão multitarefa ao MSDOS, com alguma política de escalonamento; Isto poderia fazer uso do código de reentrância ao MSDOS desenvolvido neste trabalho.

No contexto do simulador podemos citar:

- experimentar fazer o mapeamento dos discos lógicos da matriz em arquivos presentes em servidores distintos, a fim de melhorar o tempo de acesso aos dados; Isto pode ser feito reescrevendo o módulo de mapeamento para que este se comunique via rede com um *deamon* nas máquinas servidoras, cuja tarefa é servir requisições de acesso ao disco físico que ele simula;
- experimentar outras técnicas de posicionamento dos dados e de informação de redundância, via escolha do usuário no momento da carga do simulador;
- implementar uma versão do simulador que comprima os dados no momento da escrita e descomprima no momento da leitura, para economizar espaço em disco; Poderíamos utilizar algoritmos eficientes;
- implementar uma controladora de *hardware* para a matriz RAID 5 que utilize o núcleo do simulador.

Bibliografia

[Amd 67] G. M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *Proc. AFIPS 1967 Spring Joint Computer Conference*, vol 30, Apr. 1967.

[Bel 84] C. G. Bell, The Mini and Micro Industries, *IEEE Computer*, vol 17, Oct. 1984.

[Bit 88] D. Bitton & J. Gray, Disk Shadowing, *VLDB Proceedings*, Morgan Kauffman, pp. 331-338, Sep. 1988.

[Bru 59] R. K. Brunner, J. M. Harker, K. E. Haugton & A. G. Osterlund, A Gas Film Lubrication Study, Part III: Experimental Investigation of Pivoted Slider Bearings, *IBM Journal of Res. Develop.*, vol 3, pp. 260-274, 1959.

[Cab 90] L. F. Cabrera & D. D. E. Long, Swift: A Storage Architecture for Large Objects, *University of California at Santa Clara*, Tech. Report UCSC-CRL-89-04, 1990.

[Che 89] P. M. Chen, An Evaluation of Redundant Arrays of Disks using Amdahl 5890, *University of California at Berkeley*, Tech. Report UCB/CSD 89/506, May 1989.

[Dun 86] R. Duncan, Advanced MSDOS Programming, *McGrawHill*, 1986.

[Gar 84] H. Garcia-Molina, R. Cullingford, P. Honeyman & R. Lipton, The Case for Massive Memory, *Dept. of EE and CS, Princeton University*, Tech report 326, May 1984.

[Gib 89] G. A. Gibson, Performance an Reliability in Redundant Arrays of Inexpensive Disks, *University of California at Berkeley*, Sep. 1989.

[God 65] W. A. Goddard, Air Bearing Head, *U.S. Patent* 3213461, 1965.

[Gro 59] W. A. Gross, A Gas Film Lubrication Study, Part I: Some Theoretical Analyses of Slider Bearings, *IBM Journal of Res. Develop.*, vol 3, pp. 237-255, 1959.

[Har 81] J. M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana & L. G. Taft, A Quarter File Century of Disk File Innovation, *IBM Journal of Res. Develop.*, vol 25, Sept. 1981.

[Har 92] J. H. Hartman & J. K. Ousterhout, Zebra: A Striped Network File System, *University of California at Berkeley*, 1992.

[IBM 88] AS400 Programming: Backup and Recovery Guide, *IBM Form* SC21-8079-0, June 1988.

[Joh 84] O. G. Johnson, Tree-Dimensional Wave Equation Computations on Vector Computers, *Proc. IEEE*, vol. 72, Jan. 1984.

[Joy 85] , B. Joy, Presentation at ISSCC '85 panel session, Feb. 85.

[Kim 85] M. Y. Kim, Error Correcting Codes For Interleaved Disks with Minimal Redundancy, *IBM Comput. Sci. Res.*, RC11185, May 1985.

[Kim 86] M. Y. Kim, Syncronized Disk Interleaving, *IEEE Trans. on . Computers*, vol. C-35, Nov. 1986.

[Kim 86a] M. Y. Kim, Syncronous-to-Asynchronous Conversion Buffer and Aplication, *IBM Tech. Discl. Bull.*, Oct. 1986.

[Kim 87] M. Y. Kim & A. N. Tantawi, Asynchronous Disk Iterleaving: Approximating Access Delays, *IEEE Trans. on Comp.*, vol 40, no. 7, Jul. 1991.

[Lee90] E. K. Lee, Software and Performance Issues in the Implementation of a RAID Prototype, *University of California at Berkeley*, Tech. Report UCB/CSD 90/573, May 1990.

[Lee 91] E. K. Lee & R. H. Katz, Performance Consequences of Parity Placement in Disk Arrays, *ASPLOS IV*, 1991.

[Les 57] M. L. Lesser & J. W. Haanstra, The Random-Access Memory Accounting Machine - I. System Organization of the IBM 305, *IBM Journal of Res. Develop.*, vol 1, 1957.

[Lin 70] S. Lin, An Introduction to Error-Correcting Codes, *Prentice-Hall*, Englewood Cliffs, New Jersey, 1970.

[Lin 85] J. R. Lineback, New Features Tune Unix for High-end Machines, *Electronics*, Aug. 1985.

[Liv 87] M. Livny, S. Khoshafian & H. Boral, Multi-Disk Management Algorithms, *Proc. ACM SIGMETRICS*, May 1987.

[Mal 92] M. Malhotra & K. S. Trivedi, Reliability of Redundant Arrays of Inexpensive Disks (RAID), *Duke University*, 1992.

[Moo 75] G. E. Moore, Progress in Digital Integrated Electronics, *Proc. IEEE Digital Integrated Electronic Device Meeting*, 1975.

[Mye 86] G. J. Myers, A. Y. C. Yu & D. L. House, Microprocessor Technology Trends, *Proc. IEEE*, vol 74, Dec. 1986.

[Noy 57] T. Noyes & W. E. Dickinson, The Random-Access Memory Accounting Machine - II. The Magnetic-Disk, Random-Access Memory, *IBM Journal of Res. Develop.*, vol 1, 1957.

[Ous 85] J. Ousterhout et al., A Trace-Driven Analysis of the UNIX 4.2 BSD File System, *Proc. Tenth Symposium on Operating Systems Principles*, Dec. 1985.

[Ous 88] J. Ousterhout & F. Douglis, Beating the I/O Bottleneck: A Case for Log-Structured File Systems, *University of California at Berkeley*, Tech. Report UCB/CSD 88/467, Oct. 1988.

[Pat 82] A. Patel, Error and Failure-Control for a Large-Size Bubble Memory, *IEEE Trans. Magn.*, vol. MAG-18, Nov. 1982.

[Pat 88] D. Paterson, G. Gibson & R. Katz, A Case for Redundant Array of Inexpensive Disks (RAID), *Proc. ACM SIGMOD*, Jun. 1988.

[Sch 90] A. Schulman, R. J. Michels, J. Kyle, T. Paterson, D. Maxey & R. Brown, Undocumented DOS: A Programmer's Guide to Reserved MS-DOS Functions and Data Structures, *Addison-Wesley*, 1990.

[Sch 88] M. E. Schulze, Considerations in the Design of a RAID Prototype, *University of California at Berkeley*, Tech. Report UCB/CSD 88/448, Aug. 1988.

[Sie 82] D. P. Sierwiorek, C. G. Bell & A. Newell, Computer Structures: Principles and Examples, *McGraw-Hill*, 1982, p. 46.

[Sol 91] J. A. Solworth & C. U. Orji, Distorted Mirrors, *University of Illinois at Chicago*, 1991.

[Ton 90] C. L. Tondo & P. M. Adams, Writing DOS Device Drivers in C, *Prentice Hall*, Englewood Cliffs, 1990.

[Voe 55] N. A. Voegel, Air Head, *U.S. Patent* 2886651, 1955.

Apêndice A

Uma Visão Geral de Algumas Estruturas do MSDOS

Este apêndice apresenta, de modo geral, as principais estruturas internas e externas do MSDOS, documentadas e não documentadas pela Microsoft, que foram utilizadas na implementação do simulador RAID 5 apresentado no Capítulo 5. As demais estruturas e outras informações correlatas podem ser consultadas nas referências [Dun 86] e [Sch 90]. A primeira, trata das estruturas documentadas do MSDOS. A segunda, daquelas que não foram oficialmente documentadas pela Microsoft. Ambos são livros muito bons e constituem uma aquisição valiosa para quem se interessa pelo tema. As estruturas não documentadas são altamente dependentes da versão do MSDOS e não há garantias de que as novas versões do MSDOS incorporem qualquer uma delas.

Neste apêndice, só apresentaremos o formato das estruturas que são independentes da versão do MSDOS. O formato das estruturas citadas, que não forem mostrados, podem ser encontrados nas referências anteriormente mencionadas.

A.1 Introdução

Atualmente, o MSDOS possui dois tipos de sistema de arquivos. O primeiro é conhecido como FAT *filesystem*. Esse nome é originado de sua principal estrutura de dados: a tabela de alocação de arquivos – FAT (*File Allocation Table*). É utilizado, basicamente, em disco flexíveis e em discos rígidos. Além da FAT, uma outra estrutura de dados importante deste tipo de sistema de arquivos é o bloco de parâmetro de um disco – DPB (*Disk Parameter Block*).

O segundo tipo de sistema de arquivos, introduzido com o MSDOS 3.1, é conhecido como MSDOS *Network Redirector*. É utilizado para mapear a estrutura hierárquica do MSDOS em sistemas estranhos ao MSDOS, tais como servidores de rede, CD-ROMs, etc. Os discos criados com o *network redirector* não possuem FAT nem DPB. Neste apêndice, não trataremos deste tipo de sistema de arquivos.

Todos os discos, baseados em FAT ou não, possuem entradas em uma estrutura de dados chamada estrutura de diretório corrente – CDS (*Current Directory Structure*). A única exceção a essa regra são os discos criados sob a Novell NetWare.

Antes de prosseguirmos, apresentaremos como o MSDOS vê um disco físico.

A.2 O Disco Físico: Como o MSDOS o Vê

Em um disco magnético, as informações são gravadas como reversões no fluxo magnético presente em sua superfície. Este nível de abstração diz respeito apenas aos projetistas de discos magnéticos e de controladoras de disco. O MSDOS utiliza abstrações de mais alto nível. Ele apresenta ao usuário, um sistema de arquivos independente do processo de leitura e gravação dos dados.

A.2.1 Superfícies, Trilhas e Setores

Nas primeiras versões do MSDOS, o IBM-PC original vinha equipado com um acionador de disco flexível de uma face e uma cabeça, cuja capacidade de armazenamento era de $160KB$. Na superfície ativa, a cabeça escrevia e lia informação de uma das quarenta trilhas concêntricas existentes. O mecanismo de atuação da cabeça era movido entre as trilhas para posicionar a cabeça na trilha adequada. A trilha mais externa era a trilha 0 e a mais interna a 39.

Um pequeno furo, a marca de índice (*index hole*), servia como ponto de referência para determinar a rotação do disco. Um sensor gerava um pulso de índice toda vez que o furo passava por ele. Como o disco girava a uma velocidade constante de $300RPM$, a controladora podia determinar setores ao longo da trilha nos quais armazenar dados. Esses primeiros discos continham 8 setores por trilha, cada setor comportando 512 *bytes* de informação.

Algum tempo depois, os discos de uma única face foram substituídos por discos de duas faces, dobrando a capacidade de armazenamento. Um pouco antes da introdução do MSDOS 2.0, um setor extra foi adicionado a cada trilha, fazendo com que a capacidade de armazenamento aumentasse para $360KB$. Passados alguns anos, apareceram discos com maior capacidade de armazenamento, $1.2MBe1.44MB$, e em novo formato, o de 3.5".

Em todos os casos, os dados presentes no disco são identificados em termos de qual cabeça usar, de em qual trilha posicionar a cabeça e de qual setor ler ou escrever dados.

A.2.2 Número Lógico do Setor e o Conceito de Cluster

O primeiro passo no sentido de simplificar o endereço dos dados em um disco magnético (cabeça, trilha e setor) foi reconhecer que existe uma forma alternativa de especificar unicamente cada setor em um disco, usando-se apenas um único número. A forma que isso é feito é atribuir números aos diversos setores, em uma seqüência lógica. Isto é, o primeiro setor da primeira trilha sob a primeira cabeça recebe o número lógico de setor, LSN (*Logical Sector Number*), um (LSN = 1). O resto dos setores seguem a seqüência.

Um *cluster* é um grupo de setores adjacentes que são sempre tratados como unidade. Mesmo que um arquivo necessite de apenas um *byte*, ele receberá todo o *cluster*. A maioria dos formatos de discos flexíveis utiliza um *cluster* com dois setores. Os discos rígidos têm *clusters* de 4 ou 8 setores.

Para saber quais clusters pertencem a quais arquivos e quais estão livres, o MSDOS utiliza a tabela de alocação de arquivos, FAT.

A.2.3 A Tabela de Alocação de Arquivos

A FAT está sempre localizada perto do início de todo disco, logo após o setor de *boot*. Duas cópias da FAT são normalmente mantidas pelo MSDOS para o caso de falha física do disco.

A FAT é organizada como uma matriz numérica. Antes do MSDOS versão 3.0, todas as entradas na FAT tinham 12 *bits* de comprimento. Nesta versão, foi introduzida FAT de 16 *bits*.

A cada elemento da FAT corresponde um único *cluster*. Os dois primeiros elementos, referidos como *cluster* 0 e *cluster* 1, indicam o tipo do disco. O primeiro *cluster* realmente usado é o de número 2.

O *cluster* 2 é o primeiro disponível para dados. Como ambas as cópias da FAT e o diretório precedem esse espaço, o LSN do *cluster* 2 deve ser calculado pelo MSDOS utilizando os valores presentes no DPB. Desse ponto em diante, o LSN no qual um dado *cluster* inicia pode ser determinado da seguinte forma:

$$LSN(cluster_i) = (cluster_i - 2) * Sectors\ per\ cluster + LSN(cluster_2).$$

O número presente em cada elemento da FAT diz se o *cluster* correspondente está em uso ou não. E, se estiver, dá informação essencial sobre o arquivo que o está utilizando. Um zero indica que o *cluster* está livre e pode ser alocado. Os valores 1 e 2 nunca ocorrem. Os últimos 8 valores possíveis (FF8h - FFFh para FATs de 12 *bits*, ou FFF8h - FFFFh para FATs de 16 *bits*) indicam que este é o último *cluster* do arquivo. (F)FF7h indica *cluster* ruim. (F)FF0h - (F)FF6h são reservados. Qualquer outro valor indica que o arquivo que está utilizando este *cluster* é continuado no *cluster* tendo aquele valor.

Assim, a FAT contém listas ligadas que indicam a localização dos arquivos no disco. Além disso, indica as posições livres do disco. Basta se determinar o primeiro cluster de um dado arquivo, para que seja possível acessá-lo em sua integridade. Esta informação vem do diretório do disco.

A.2.4 A Estrutura do Diretório

Todo disco tem um diretório raiz (*root directory*), o qual é o ponto de partida no processo de tradução de nomes de arquivos em uma seqüência de *clusters*.

O diretório raiz segue imediatamente a FAT e precede a área de dados. Seu tamanho é estabelecido quando o disco é formatado e, ao contrário dos diretórios não raiz (que são implementados como arquivos), não pode mudar.

Cada entrada no diretório, não importa se é o diretório raiz ou um subdiretório, consiste da seguinte estrutura de dados:

```
struct DirItem {
        char          FileName[8];
        char          Ext[3];
        unsigned char Attribute;
        char          Unused[10];
        unsigned int  Time;
        unsigned int  Date;
        unsigned int  FirstCluster;
        unsigned long Size;
};
```

Como dissemos anteriormente, a entrada no diretório correspondente a um arquivo indica o primeiro *cluster* utilizado por este arquivo.

A.3 A Lista das Listas

Desde a introdução do `CONFIG.SYS` com o MSDOS 2.0, uma coleção de apontadores, denominada lista das listas – LOL (*List of Lists*), vêm sendo mantida perto do início do segmento de dados do núcleo do MSDOS.

A LOL é a espinha dorsal do MSDOS. Algumas das estruturas que podem ser acessadas a partir da LOL são:

DOS List of Lists
Utility Functions
Memory Control Block (MCB)
    Program Segment Prefix (PSP)
        Environment Segment
        File Handle Table
    DOS 4.x Data Segment Subsegment Control Blocks
        STACKS Segments
Disk Parameter Block (DPB)
    File Allocation Table (FAT)
System File Table (SFT)
Device Driver Chain
DiskBuffers
Current Directory Structure (CDS)
    Installable File System (IFS) *Record*
FCB *Table*
`SHARE.EXE` *Hooks*
    Sharing Record
    Lock Record

A.3.1 Como a LOL é Composta

A composição da LOL tem variado, às vezes significativamente, de uma versão para a outra do MSDOS. Antes da introdução do suporte à rede no MSDOS 3.1, esta estrutura de dados variava muito. Contudo, uma vez que o suporte à rede se tornou disponível, estabilidade foi forçada nesta estrutura. Sem isto, os programas de rede teriam dificuldade em operar.

Um apontador FAR (apontador de 32 *bits*) para o décimo terceiro *byte* a partir da LOL é retornado por `INT 21h` (*dispatcher* do MSDOS), subfunção 52h, em `ES:BX` (registradores da família de processadores 80x86).

Os principais componentes da LOL são:

- apontador FAR para o *buffer* de disco correntemente em uso;
- apontador FAR para o primeiro DPB; Cada DPB se liga ao próximo, formando uma cadeia que pode ser usada para acessar o DPB de qualquer disco presente no sistema; Cada DPB também pode ser acessado via um apontador FAR na CDS do disco correspondente; `INT 21h`, subfunção 32h, retorna um apontador FAR em `DS:BX` para um dado DPB;
- apontador FAR para a SFT (*system file table*), a qual contém informações sobre arquivos e sobre *device drivers* que são acessados via *handles*;

- *word* que contém o maior número setores/*buffer* dentre todos os *device drivers* de bloco do sistema;
- apontador FAR para informações do *buffer* de disco;
- apontador FAR para uma matriz de CDSs; Cada disco no sistema tem sua própria CDS, a qual contém o *path* e aponta para o DPB daquele disco; Esta estrutura também contém *bits* de atributo que especificam se o disco existe ou não, se foi modificado pelos comandos `JOIN`, `ASSIGN` e `SUBST`, e se é um *network drive*;
- apontador FAR para a tabela de FCBs (*File Control Blocks*);
- *byte* indicando o número total de dispositivos de bloco instalados no sistema;
- *byte* contendo o valor ajustado pelo comando `LASTDRIVE` no `CONFIG.SYS` (o valor padrão é 5);
- *byte* mostrando o número de discos `JOIN`ed.

A.3.2 A Construção da LOL

A LOL é uma estrutura de dados dinâmica que reflete as mudança feitas no `CONFIG.SYS`, e as feitas por comandos que mudam a identidade dos vários discos e diretórios enquanto o sistema está em operação. Por esta razão, a LOL é construída no momento da inicialização do sistema.

A primeira coisa que acontece quando o sistema é inicializado é a leitura do *boot sector* do disco flexível no acionador `A:`, se houver algum presente. Se não, ele lê o *boot sector* do disco rígido, se possível. Se nenhuma leitura for bem sucedida, o sistema não pode ser inicializado.

Quando o *boot sector* é lido na RAM, o código que ele contém localiza o arquivo `IO.SYS` (ou o arquivo `IBMBIO.COM`), que sempre começa no *cluster* 2 do disco de *boot*, o carrega para a memória, e transfere o controle para o código de inicialização deste arquivo. Este código faz muitas coisas: move a si mesmo para o topo da memória, carrega o arquivo `MSDOS.SYS` (ou o arquivo `IBMDOS.COM`), processa o `CONFIG.SYS`, usa a informação contida ali para montar a LOL , e, finalmente, dispara o `COMMAND.COM`.

A.4 O Bloco de Parâmetros de Disco

Para cada disco do sistema, existe um bloco de parâmetros de disco – DPB (*Disk Parameter Block*). Este bloco contém informações que o MSDOS usa para converter números de *clusters* em LSNs e para associar o dispositivo com uma letra identificadora (por exemplo, `A:`).

O DPB para cada disco é criado imediatamente após o MSDOS chamar a rotina de inicialização do disco durante o processo de *boot*. Aqueles discos que estão presentes em `IO.SYS`, normalmente `A:`, `B:` e `C:`, são criados na inicialização de `IO.SYS` antes do processamento do `CONFIG.SYS`. Os DPBs dos outros discos são criados como um dos últimos passos na instalação do *device driver* associado, quando do processamento do `CONFIG.SYS`.

Para criar o DPB de um dado disco, usa-se `INT 21h`, subfunção `53h`, passando a ela um apontador FAR para o bloco de parâmetros do BIOS – BPB (*BIOS Parameter Block*) – do disco em questão.

Cada DPB é ligado ao próximo via um apontador FAR presente no DPB. O fim da lista é indicado por um apontador FAR cujo *offset* vale FFFFh.

Embora os DPBs estejam ligados em uma lista cujo início está presente na LOL, a melhor forma de obter o DPB de um dado disco é através de `INT 21h`, subfunção 32h.

A.5 Tabelas de Arquivos do Sistema e Tabela de Arquivos da Tarefa

Enquanto as DPBs relacionam os discos físicos com as letras que o MSDOS usa para referenciar esses dispositivos, as tabelas de arquivos do sistema – SFT (*System File Tables*) – formam a espinha dorsal do sistema de arquivos do MSDOS, tendo sido introduzidas na versão 2.0.

As SFTs mantêm o status de todos os arquivos abertos. Isto inclui associação de nomes a uma entrada do diretório, deslocamento dentro do arquivo, tamanho do arquivo, etc. Toda a informação contida na entrada do diretório para um dado arquivo vem da SFT.

O número de entradas na cadeia de SFTs é estabelecido pelo parâmetro `FILES` do `CONFIG.SYS`. Se este parâmetro não for definido, o conjunto de SFTs possuirá oito entradas. Todos os *handles* de arquivo que um programa manipula vêm de alguma entrada da lista de SFTs.

Quando algum aplicativo pede ao MSDOS para abrir um arquivo, as seguintes ações são tomadas pelo MSDOS.

Primeiro o MSDOS procura na tabela de *handles* do aplicativo, localizada em seu PSP, e também chamada tabela de arquivos da tarefa – JFT (*Job File Table*), até encontrar uma entrada não utilizada. Se tudo correr bem, este índice se tornará o *handle* associado ao arquivo. Se uma entrada livre não for encontrada, o arquivo não poderá ser aberto.

Em seguida, a cadeia de SFTs é varrida, procurando-se pela primeira entrada que tenha uma contagem de *handles* igual a zero. Se não for encontrada, o MSDOS exibe a mensagem `Out of Handles` e a requisição falha. Se for encontrada, o MSDOS acessa o diretório raiz do disco especificado utilizando sua CDS. Em seguida, percorre a árvore de diretórios até encontrar o diretório referenciado. Se não encontrar, a requisição falha. Neste ponto, o MSDOS vê se o arquivo referenciado é um arquivo real ou se é um *device driver* de caracter tal como `LPT1`. Isto significa que todos os *device drivers* de caracter parecem existir em todos os diretórios. Também significa que qualquer arquivo que tenha o nome de um *device driver* de caracter nunca será aberto, não importando sua extensão. Se não for achado um *device driver* com aquele nome, então o último diretório é varrido em busca do arquivo especificado. Se este não for encontrado, a requisição falha, retornando a mensagem `Path not Found`.

Se for achado um *device driver* de caracter ou um arquivo, o contador de *handles* da entrada correspondente na SFT é incrementado, e o índice dessa entrada é colocada na JFT do programa.

Para um arquivo, todas as informações pertinentes, contidas no diretório, são copiadas para a SFT. O marcador de deslocamento dentro do arquivo é colocado em zero para indicar que nada foi lido do arquivo. Já, para um *device driver*, alguns *bits* de controle na entrada correspondente da SFT são ajustados para controlar as funções de entrada e saída. Além disso, um apontador FAR para o código do *device driver* é armazenado na SFT. Em ambas as situações, o MSDOS retorna um *handle* que deve ser usado em todas as referências subseqüentes ao arquivo. Este *handle* é simplesmente um índice na JFT, e o *byte* naquele índice (JFT[*handle*]) é um índice para a lista de SFTs.

Se estiver sendo criado um novo arquivo, ao invés de apenas abrir um que já existe, ocorre esta mesma seqüência de eventos, com uma exceção: a mensagem de erro `Path not found` só é gerada se algum diretório no *path* não puder ser localizado.

Quando um arquivo é fechado, a SFT é acessada da mesma forma que para leituras e escritas. A entrada no diretório é atualizada com informações da SFT, e as escritas pendentes para aquele arquivo são executadas imediatamente. O contador de *handles* na entrada da SFT é decrementado, e o índice da SFT na JFT é reposto pelo valor FFh para indicar que a entrada na JFT está disponível. Quando o contador de *handles* na SFT chegar a zero, esta entrada da SFT estará liberada para ser usada novamente[1].

A.6 Estrutura de Diretório Corrente

A matriz de estruturas de diretório corrente – CDS (*Current Directory Structure*) – é uma estrutura de dados de fundamental importância no sistema de arquivos do MSDOS. Ela foi introduzida na versão 3.0.

A CDS foi adotada como parte da adição do suporte à redes feita no MSDOS, e possui um papel central na manipulação de sistemas de arquivos não incluídos no próprio MSDOS.

A matriz de CDSs contém um CDS para cada disco possível de existir no sistema. Assim, se for especificado `LASTDRIVE = Z`, existirá uma matriz de CDSs com 26 entradas. Como o tamanho dessa matriz é fixado por `LASTDRIVE`, normalmente ela não pode ser expandida sem alteração no `CONFIG.SYS` e sem reinicializar a máquina.

A CDS contém informações importantes, tais como: o *path* corrente no disco, indicador do tipo do disco (`NETWORK`, `PHYSICAL`, `JOIN` ou `SUBST`), apontador FAR para o DPB do disco, etc.

A LOL contém um apontador FAR para a matriz de CDSs.

A.7 A Área de Dados do MSDOS

A área de dados do MSDOS – SDA (*Swappable Data Area*) – pode ser obtida via `INT 21h`, subfunção `5D06h` para as versões 3.1 a 3.3, 5.0 e 6.0, e via `Int 21h`, subfunção `5D0Bh` para as versões 4.x. Esta área contém o contexto corrente do MSDOS, que inclui o segmento do PSP corrente, as três pilhas internas do MSDOS, etc.

O conhecimento desta área nos permitiu implementar a reentrância ao MSDOS do simulador RAID 5. Como um dos passos necessários para se reentrar o MSDOS, podemos salvar e restaurar a SDA. Este procedimento nos permite chamar o MSDOS, quase sempre, sem ter que esperar que os *flags* do MSDOS indiquem que ele pode ser reentrado. Em nosso simulador, esta é uma característica crucial, pois dentro do código de um *device driver*, esses *flags* sempre indicam que o MSDOS está ocupado.

As funções mencionadas acima são usadas em conjunto com `INT 2Ah`. Quando `INT 2Ah`, subfunção `80h`, é invocada, temos uma indicação de que o MSDOS está em uma região crítica. Quando o MSDOS está em uma região crítica, a SDA não pode ser movida de lugar. O fim de uma seção crítica é indicada por uma chamada a `INT 2Ah`, subfunções `81h` ou `82h`.

`INT 21h`, subfunção `5D06h`, retorna a seguinte informação:

[1] Este contador é maior que um se o arquivo foi aberto mais de uma vez por um mesmo programa ou por programas diferentes.

```
DS:SI - apontador para a SDA
DX    - tamanho da area a salvar quando INDOS > 0
CX    - tamanho da area que sempre deve ser salva
```

INT 21h, subfunção 5D0Bh, retorna em DS:SI um apontador FAR para uma lista de SDAs, que contém:

```
Deslocamento    Tamanho    Descrição
------------    -------    -----------
00h             WORD       quantidade de SDAs
SDA_ENTRY:
02h             DWORD      endereço desta SDA
06h             WORD       tamanho e tipo da area de dados
                           bit 15      - 1 se salvar sempre
                                       - 0 se salvar quando INDOS > 0
                           bits 0..14 - comprimento em bytes
SDA_ENTRY:
.
.
.
```

A.8 Prefixo de Segmento de um Programa

A compreensão total do prefixo de segmento de um programa – PSP (*Program Segment Prefix*) – é vital para uma programação com sucesso no MSDOS. É uma área reservada, de 256 *bytes* de comprimento, instalada pelo MSDOS no início do bloco de memória alocado para um programa transitório. O PSP contém alguns encadeamentos ao MSDOS que podem ser utilizados pelo programa transitório, alguma informação que o MSDOS grava para seu próprio propósito e alguma informação que o MSDOS passa ao programa transitório.

O PSP consiste da seguinte estrutura de dados:

```
FINI:       INT     20h             ; 0000 saida tipo CP/M
NXTGRAF     DW      0A000h          ; 0002 primeiro segmento livre
            DB      0               ; 0004 byte para alinhamento
CPMCAL:     CALLF   INT21DSP        ; 0005 chamada de serviço tipo CP/M
ISV22       DD      0               ; 000A vetor de termino
ISV23       DD      0               ; 000E vetor de CTRL-C
ISV24       DD      0               ; 0012 vetor de erro critico
PARENT      DW      PARENT_ID       ; 0016 PSP do processo pai
HANDLES     DB      1, 1, 1, 0, 2   ; 0018 indices na SFT
            DB      15 DUP (255)
ENVPTR      DW      ENVIRON         ; 002C segmento do ambiente
SAVSTK      DD      0               ; 002E SS:SP salvo por Int 21h
NHDLS       DW      20              ; 0032 numero de handles disponiveis
HTBLPTR     DD      HANDLES         ; 0034 apontador para a tabela de handles
            DD      -1              ; 0038 PSP anterior do SHARE
```

```
RSVD1       DB      14 DUP (0)      ; 003C sem uso
DISP:       INT     21h             ; 0050 dispatcher tipo UNIX
            RETF
RSV2        DB      9 DUP (0)       ; 0053 sem uso
FCB1        DB      0,' '           ; 005C FCB1
            DB      0, 0, 0, 0
FCB2        DB      0,' '           ; 006C FCB2
            DB      0, 0, 0, 0
TAILC       DB      5               ; 0080 contador do command tail
TAIL        DB      ' args'         ; 0081 command tail
            DB      0Dh
            DB      127 DUP (0)
```

Os campos do PSP diretamente manipulados pela implementação do simulador RAID 5 são:

- ENVPTR: contém o segmento do bloco de ambiente, que contém uma série de *strings* terminadas por zero; O bloco de ambiente é herdado do programa que chamou EXEC para carregar o programa atualmente em execução; Ele contém informações como o *path* de pesquisa atual utilizado pelo COMMAND.COM para encontrar programas executáveis, a posição no disco do próprio COMMAND.COM, etc;

- TAILC: contém o restante da linha de comando que acionou o programa transitório, após o nome do programa; O comprimento de TAILC, não incluindo o caracter de retorno em sua extremidade, é colocado no *byte* com *offset* 80h; Os parâmetros de redirecionamento ou de conexão e seus nomes de arquivos associados não aparecem em TAILC pois o redirecionamento é transparente às aplicações; TAILC também serve como área de transferência de disco – DTA (*Disk Transfer Address*) – padrão, definida pelo MSDOS antes de passar o controle ao programa transitório; Se o programa não muda a DTA explicitamente, quaisquer operações de leitura ou escrita de arquivo, solicitadas com o grupo de chamadas de função do FCB, utilizam automaticamente esta área como *buffer* de dados.

A.9 Device Drivers

Os *device drivers* são módulos de um sistema operacional que controlam o *hardware*. Eles isolam o núcleo do sistema operacional das características e idiossincrasias específicas dos dispositivos periféricos que fazem interface com o processador central.

Os *device drivers*[2] para o MSDOS se encaixam em duas classes distintas:

1. *device drivers* de bloco;

2. *device drivers* de caracter.

A classe de um *device driver* determina que funções deve suportar, como é visto pelo MSDOS e de que modo faz o dispositivo associado se comportar quando um programa aplicativo faz uma solicitação de *I/O*.

Neste apêndice, só trataremos de *device drivers* de bloco.

[2]Os *device drivers* instaláveis foram introduzidos na versão 2.0 do MSDOS.

A.9.1 Device Drivers de Bloco

Os *Device drivers* de bloco normalmente controlam dispositivos de armazenamento em massa de acesso randômico, tais como discos magnéticos. Eles também podem ser usados para controlar dispositivos de acesso não randômico tais como fita magnética. Os *device drivers* de bloco transferem dados em blocos, ao invés de um *byte* por vez. O tamanho dos blocos pode ser fixo (discos) ou variável (fita).

Um *device driver* de bloco pode suportar uma unidade de *hardware*, mapear uma única unidade física em duas ou mais unidades lógicas, ou ambas. Os *device drivers* de bloco não possuem nomes lógicos do tipo arquivo, como fazem os *device drivers* de caracter (por exemplo, `CON`). Em vez disto, o MSDOS atribui designadores alfabéticos às unidades de *device drivers* de bloco ou às unidades lógicas, em ordem crescente: `A:`, `B:`, etc. Cada unidade lógica contém um sistema de arquivos.

A posição do *device driver* de bloco na cadeia de todos os *device drivers* determina a primeira letra atribuída a ele. O número de unidades de acionamento lógico suportado determina o número de letras atribuídas.

A.9.2 Estrutura de um Device Driver de Bloco

Um *device driver* de bloco consiste de três partes principais:

1. cabeçalho;
2. rotina de estratégia (*strat*);
3. rotina de interrupção (*intr*).

Essas três partes estão apresentadas na Figura A.1.

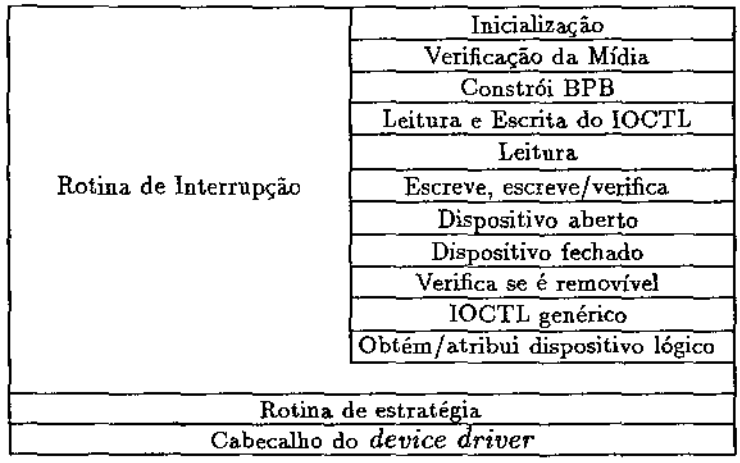

Figura A.1: Estrutura Geral de um *Device Driver* de Bloco

O Cabeçalho

O cabeçalho do *device driver* de bloco, definido por:

```
struct BlockDeviceHeader {
        struct DeviceHeader far* Link;
        unsigned int             Attributes;
        (near*                   Strategy) (void);
        (near*                   Interrupt) (void);
        unsigned char            BlockDevices;
        unsigned char            Dummy[7];
};
```

está localizado no *offset* zero a partir do início de seu código. Contém um *link* ao próximo *device driver* na cadeia, um conjunto de *flags* de atributo, *offsets* relativos ao segmento de código do *device driver* do código de estratégia e do código de interrupção, e o número de unidades lógicas suportado.

A Tabela A.1 mostra o significado dos diversos *flags* de atributo.

Bit	Significado
15	0, indicando *device driver* de bloco
14	1 se IOCTL_WRITE e IOCTL_READ forem suportadas
13	1 se o BPB for utilizado para determinar as características da mídia 0 se o *byte MediaID* na FAT for utilizado
12	Reservado (deve ser 0)
11	1 se suporta mídia aberta/fechada/removível (MSDOS 3.0+)
7-10	Reservado (deve ser 0)
6	1 se IOCTL genérico suportado
5	Reservado (deve ser 0)
2-4	0, indicando *device driver de bloco*
1	1 se o *device driver* suporta endereçamento de 32 *bits* (MSDOS 4.0+)
0	0, indicando *device driver* de bloco

Tabela A.1: Palavra de Atributo de um *Device Driver* de Bloco

A Rotina de Estratégia

O MSDOS chama a rotina de estratégia (*strat*) para o *device driver* quando este é instalado e, novamente, sempre que um programa aplicativo emitir uma solicitação de *I/O* relativo ao dispositivo que o *device driver* controla. O MSDOS passa à rotina de estratégia um apontador FAR para uma estrutura de dados chamada cabeçalho da solicitação (*request header*). Esta estrutura contém informações à respeito do tipo de operação a ser executada.

Os primeiros 13 *bytes* do *request header* são os mesmos para todas as funções relativas ao *device driver*, e são, freqüentemente, referidos como sendo a parte estática do *request header*:

```
struct RH_StaticPart {
```

```
        unsigned char RequestHeaderLength;
        unsigned char UnitNumberForThisRequest;
        unsigned char CommandCode;
        unsigned int  ReturnedStatus;
        unsigned char Reserved[8];
};
```

Eles indicam o comprimento em *bytes* do *request header*, o número da unidade lógica à qual a requisição é dirigida, o código do comando requisitado e o *status* da requisição. O número e o conteúdo dos *bytes* subseqüentes variam de acordo com o tipo de função que está sendo solicitada. Tanto o MSDOS como o *device driver* lêem e escrevem no *request header*.

A Tabela A.2 mostra o significado dos bits da palavra de *status*, e a Tabela A.3 os códigos de erro devolvidos nos *bits* 0 a 7 da palavra de *status*.

Bits	Significado
15	Erro
12-14	Reservado
9	Ocupado
8	Realizado
0-7	Código de erro se *bit* 15 = 1

Tabela A.2: Palavra de *Status* de um *Device Driver* de Bloco

Código	Significado
0	Violação da proteção contra escrita
1	Unidade desconhecida
2	Unidade não está pronta
3	Comando desconhecido
4	Erro de dados (CRC)
5	Erro no tamanho do *request header*
6	Erro de *seek*
7	Mídia desconhecida
8	Setor não encontrado
9	Impressora sem papel
10	Erro de escrita
11	Erro de leitura
12	Falha geral
13-14	Reservado
15	Mudança inválida do disco (MSDOS 3.0+)

Tabela A.3: Códigos de Erro Devolvidos nos *Bits* 0 a 7 da Palavra de *status* no *Request Header*.

A Rotina de Interrupção

A parte mais complexa de um *device driver* é a rotina de interrupção (*intr*), que o MSDOS chama imediatamente após chamar a rotina de estratégia. A rotina de interrupção executa as operações de *I/O* reais, baseando-se nas informações passadas no *request header*.

Quando uma função de *I/O* se completa, a rotina de interrupção utiliza o campo de *status* no *request header* para informar ao núcleo do MSDOS a respeito do resultado da operação solicitada. Outros campos no *request header* são usados para devolver informações úteis, tal como contagem dos setores realmente transferidos.

Normalmente, a rotina de interrupção consiste nos seguintes elementos:

- uma coletânea de subrotinas para implementar os diversos tipos de funções que podem ser solicitados pelo MSDOS;
- um ponto de entrada, centralizado, que grava todos os registradores afetados, extrai o código de função desejado a partir do *request header* e desvia para a rotina apropriada;
- um ponto de saída centralizado que armazena códigos de *status* e de erros no *request header*, e restaura o conteúdo anterior dos registradores afetados.

Apesar de sugerido pelo nome, a rotina de interrupção nunca é chamada de forma assíncrona.

A.9.3 Os Comandos de um Device Driver de Bloco

São definidos um total de 14 códigos de comando para os *device drivers* de bloco sob o MSDOS. Os códigos de comando, os nomes das rotinas e as versões do MSDOS em que são suportadas pela primeira vez, são mostradas na Tabela A.4.

Código	Função	Versão
0	Inicialização	2.0
1	Verificação da mídia	2.0
2	Constrói BPB	2.0
3	Leitura IOCTL	2.0
4	Leitura	2.0
8	Escrita	2.0
9	Escrita com verificação	2.0
12	Escrita em IOCTL	2.0
13	Dispositivo aberto	3.0
14	Dispositivo fechado	3.0
15	Mídia removível	3.0
19	IOCTL genérico	3.2
23	Obtém dispositivo lógico	3.2
24	Atribui dispositivo lógico	3.2

Tabela A.4: Comandos Relativos a um *Device Driver* de Bloco

O simulador da Arquitetura RAID 5, apresentado no Capítulo 5, utiliza as seguintes funções:

- Inicialização (`Init`);
- Verificação da mídia (`MediaCheck`);
- Constrói BPB (`BuildBPB`);
- Leitura (`Read`);
- Escrita (`Write`);
- Escrita com verificação (`WriteWithVerify`);
- Escrita em IOCTL (`IOCTLWrite`).

A definição da interface dessas funções é bastante extensa e detalhada. Está fora do escopo deste apêndice apresentá-las. Contudo, sugerimos a referência [Dun 86] como elemento de consulta.

A.9.4 Processamento de uma Solicitação Típica de I/O

Um programa aplicativo solicita uma operação de *I/O* a partir do MSDOS carregando registradores com valores apropriados e executando uma `INT 21h`. Isto resulta na seguinte seqüência de ações:

1. o MSDOS inspeciona suas tabelas internas e determina qual *device driver* de bloco deve receber a solicitação;
2. o MSDOS cria um pacote de dados (*request header*) em uma área reservada da memória;
3. o MSDOS chama o ponto de entrada (*strat*) do *device driver*, passando o endereço do *request header* em `ES:BX`;
4. o *device driver* salva o endereço do *request header* em uma variável local e executa um retorno tipo FAR;
5. o MSDOS chama o ponto de entrada (*intr*) do *device driver*[3];
6. a rotina de interrupção salva todos os registradores, recupera o endereço do *request header* que havia sido salva pela rotina de estratégia, extrai o código da função e desvia para a subrotina apropriada;
7. caso tenha sido solicitado uma transferência de dados em um dispositivo de bloco, a subrotina de leitura ou de escrita do *device driver* traduzirá o número do setor lógico em um endereço de cabeça, trilha e setor físico para a unidade solicitada, e depois realiza a operação;
8. quando a solicitação for completada, a rotina de interrupção estabelece a palavra de *status* e qualquer outra informação solicitada no *request header*, restaura todos os registradores a seu estado de entrada e executa um retorno tipo FAR;
9. o MSDOS traduz o *status* de retorno do *device driver* ao código de retorno apropriado e o valor do *flag* `CARRY` para a função da `INT 21h` do MSDOS que foi solicitada, e devolve o controle ao programa aplicativo.

[3] O MSDOS divide a requisição em dois passos (*strat* e *intr*) para manter compatibilidade com o OS/2. Neste sistema operacional, *strat* é utilizado para organizar os acessos físicos ao disco.