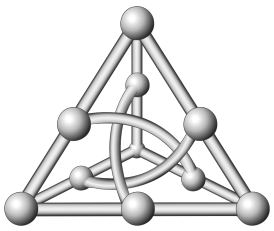
Basic Solution File System

Vitor de Freitas Benites
Gustavo Lopez Flores
Fabio Batista Rodrigues

Monografia

Área de Concentração: Sistemas Operacionais

Orientador: Prof. Brivaldo Alves da Silva Jr.



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
Dez, 2025

Basic Solution File System

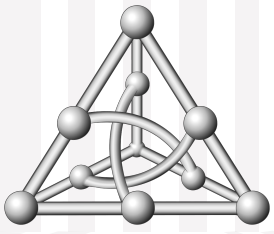
Vitor de Freitas Benites
Gustavo Lopez Flores
Fabio Batista Rodrigues

Monografia

Área de Concentração: Sistemas Operacionais

Orientador: Prof. Brivaldo Alves da Silva Jr.

Apresentado em cumprimento parcial dos requisitos para o grau de bacharel em Sistemas de Informação



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
Dez, 2025

Agradecimentos

Eu, Vitor de Freitas Benites, agradeço à minha família pelo apoio contínuo durante todo o período de estudo e desenvolvimento deste projeto. Sua compreensão, incentivo e presença constante forneceram as condições necessárias para que eu pudesse me dedicar a esta pesquisa com tranquilidade.

Agradeço ao Professor Dr. Brivaldo Alves da Silva Jr. por sua orientação ao longo de todo o desenvolvimento deste trabalho. As discussões, sugestões e a oportunidade de aprofundar conhecimentos durante as reuniões foram essenciais para a construção deste projeto e para o meu crescimento acadêmico na área de computação.

Agradeço aos colegas de grupo, cuja colaboração e troca de ideias foram essenciais para transformar o *Basic Solution File System* em um projeto viável. O apoio, as discussões e as contribuições técnicas enriqueceram o desenvolvimento deste trabalho e tornaram possível sua realização.

Eu, Gustavo Lopez Flores, agradeço aos meus familiares, pelo incentivo e apoio essenciais para a realização do ensino superior na capital. Foi graças à confiança que depositaram em mim, especialmente quando optei por seguir nesta área e me mudar para outra cidade, que encontrei forças para continuar. Deixo um agradecimento especial à minha avó Erotildes, minha irmã Erika e ao meu tio Elcio, por todo apoio até o final do curso.

Aos meus colegas, Fabio Batista Rodrigues e Vitor de Freitas Benites, minha gratidão por aceitarem o desafio de realizar este projeto em equipe e por terem me acolhido. Foi uma honra contar com a colaboração e aprender com ambos ao longo desta jornada. O sucesso do projeto *Basic Solution File System* é um testemunho direto da dedicação e união do grupo, que transformou um trabalho acadêmico em uma valiosa experiência de crescimento pessoal e profissional.

Ao Professor Dr. Brivaldo Alves da Silva Jr., agradeço por ter proposto este projeto e pela oportunidade de trabalharmos juntos. Sinto-me honrado por poder aprofundar meus conhecimentos na área de Sistemas Operacionais e Infraestrutura sob a orientação de uma referência na área. Agradeço imensamente pelo tempo dedicado, pelas muitas semanas de reuniões e, sobretudo, pela confiança depositada em nossa equipe.

Eu, Fabio Batista Rodrigues, agradeço aos meus pais, em especial minha mãe, Luzia Batista e meu pai, Raul Sergio, desde o inicio incentivando esforço e trabalho duro em busca de meus objetivos.

Agradeço à minha namorada, Giovana da Silva Menas Mühl, que, nessa cami-

nhada e especialmente nos últimos anos da graduação, esteve ao meu lado em cada passo. Foi ela quem me apoiou quando mais precisei, oferecendo força emocional e me ajudando a seguir em frente nos momentos mais desafiadores, e quem comemorou comigo cada conquista. Sua presença constante nos momentos difíceis e felizes tornou essa jornada muito mais leve e significativa.

Agradeço ao meu amigo e colega neste trabalho, Vitor de Freitas Benites, pelo companheirismo e pelos momentos compartilhados dentro e além da graduação. Sem conhecê-lo, eu não teria vivido experiências que ultrapassam o ambiente acadêmico. Levarei todas elas comigo para sempre. Ao colega Gustavo Lopez Flores, agradeço por dividir o aprendizado que construímos neste trabalho.

Agradeço também ao triângulo (*professor – aluno – conhecimento*) do professor Dr. Brivaldo Alves da Silva Jr, pois foi por meio dele que conheci a beleza dos Sistemas Distribuídos e de Sistemas Operacionais e, acima de tudo, aprendi a me desafiar. Foi por meio disso que descobri um mundo à parte, capaz de ampliar meus horizontes dentro do universo da computação. Obviamente, sou grato pelo seu exemplo de profissionalismo e dedicação, valores que levarei comigo por toda a vida.

Por fim, o grupo agradece a todos os professores e funcionários da FACOM, que de alguma forma contribuíram significativamente para nossa formação. Somos igualmente gratos aos membros da banca, professores Irineu Sotoma, Renan Albuquerque Marks e Samuel Benjoino Ferraz pela disponibilidade em participar da avaliação deste trabalho. Somos gratos à Universidade Federal de Mato Grosso do Sul por proporcionar um ambiente qualificado de ensino, pesquisa e extensão.

Resumo

A falta de mecanismos nativos para a recuperação estruturada de arquivos excluídos em sistemas de arquivos amplamente utilizados constitui uma limitação ainda não solucionada. Nesse sentido, este trabalho apresenta o desenvolvimento e a avaliação do *Basic Solution File System* (BSFS), um sistema de arquivos experimental implementado em linguagem C e executado em espaço de usuário no Linux, com um modelo próprio de recuperação de arquivos excluídos. O estudo é fundamentado em uma análise comparativa das arquiteturas e práticas de implementação utilizadas em sistemas de arquivos como Ext2, Ext3, Ext4, ZFS e Btrfs, com ênfase nas estratégias de gerenciamento e otimização da alocação de blocos, organização de metadados e métodos de preservação de integridade. São descritas a organização interna do BSFS, suas estruturas de dados, os utilitários de formatação e acesso, bem como os testes comparativos de desempenho efetuados com sistemas amplamente utilizados no Linux como Ext4, Btrfs e ZFS. Os resultados mostram que o BSFS é funcional e capaz de recuperar arquivos excluídos sem o uso de ferramentas externas ou técnicas complexas de recuperação de *i-nodes*. O trabalho contribui com uma implementação prática que permite analisar, de forma controlada, os impactos de diferentes escolhas de projeto na consistência, recuperação e gerenciamento de blocos em sistemas de arquivos.

Palavra-chave: sistema de arquivos, linux, recuperação de dados.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

> *I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*
>
> – Linus Torvalds

Os sistemas de arquivos constituem um dos componentes fundamentais de um sistema operacional, sendo responsáveis pela organização lógica, armazenamento e acesso aos dados em dispositivos de memória secundária. Sua função é oferecer uma abstração estruturada sobre o *hardware* de armazenamento, permitindo que programas e usuários interajam com os dados de forma hierárquica, segura e eficiente. A confiabilidade e o desempenho das estruturas de dados que compõem um sistema de arquivos afeta diretamente a estabilidade e a experiência de uso, especialmente em ambientes que lidam com grandes volumes de informação.

Com o avanço das tecnologias de armazenamento e a popularização de unidades de estado sólido (*solid-state drives* — SSDs), surgiram novos desafios na concepção de sistemas de arquivos [12, 20]. Entre eles estão a necessidade de otimizar o acesso à dados não contíguos, reduzir o número de escritas desnecessárias e oferecer mecanismos de recuperação capazes de preservar a integridade das informações em caso de falhas. Esses aspectos motivaram o surgimento de sistemas como o Ext4, Btrfs e ZFS, que introduziram técnicas avançadas de alocação, *journaling* e *copy-on-write*, representando marcos na evolução do gerenciamento de armazenamento moderno [8].

O estudo e a implementação de sistemas de arquivos continuam sendo uma área de interesse na pesquisa em sistemas operacionais, tanto pelo seu valor didático quanto pelo potencial de inovação em novos modelos de gerenciamento de dados. A criação de implementações experimentais permite investigar soluções alternativas para problemas clássicos, como a fragmentação e o gerenciamento de blocos livres,

favorecendo o entendimento detalhado de seus impactos sobre o desempenho e a consistência dos dados.

Apesar dos avanços presentes nos sistemas de arquivos modernos, a maioria deles ainda carece de um mecanismo direto e integrado para recuperação de arquivos excluídos. Em implementações amplamente utilizadas, como o Ext4, Btrfs e ZFS, a exclusão de um arquivo implica a remoção imediata de suas referências nos metadados, tornando sua restauração dependente de ferramentas externas, que operam de forma limitada e nem sempre confiável. Esses utilitários, em geral, realizam varreduras heurísticas sobre os blocos de dados, sem garantia de integridade ou consistência estrutural, o que restringe seu uso a situações emergenciais e a casos simples de perda recente.

Ferramentas de nível de usuário que implementam uma lixeira, como `trash-cli` ou mecanismos equivalentes presentes em ambientes gráficos, não constituem um método de recuperação propriamente dito, apenas transferem o arquivo para um diretório especial antes que a operação de exclusão seja efetivada pelo sistema de arquivos. Uma vez removidas as referências do *i-node*, tais ferramentas deixam de ser úteis, pois não atuam sobre as estruturas internas nem preservam informações necessárias para a restauração posterior. Assim, esses mecanismos oferecem conveniência ao usuário, mas não substituem soluções integradas ao sistema de arquivos, capazes de operar de forma estruturada após a exclusão real dos dados.

Essa limitação evidencia uma lacuna entre o gerenciamento eficiente de armazenamento e a preservação lógica de dados em nível de sistema de arquivos. Embora a integridade seja tratada por meio de técnicas como *journaling* e *copy-on-write*, os sistemas não se preocupam com a possibilidade de recuperação reversível de exclusões lógicas realizadas pelo usuário. A ausência desse tipo de funcionalidade torna a recuperação de arquivos um processo dependente do contexto de uso e das ferramentas disponíveis, sem padronização ou garantias sobre o estado final dos dados recuperados.

Portanto, torna-se relevante a investigação e o desenvolvimento de abordagens que permitam integrar ao próprio sistema de arquivos mecanismos internos de recuperação, capazes de registrar, preservar e restaurar arquivos excluídos de forma controlada, sem comprometer a consistência das estruturas e sem recorrer a soluções externas.

1.1 Objetivo

O objetivo geral deste trabalho é desenvolver e avaliar um sistema de arquivos experimental em espaço de usuário denominado *Basic Solution File System* (BSFS), implementado em linguagem C. O sistema tem como propósito investigar a viabilidade de um modelo simples de gerenciamento e recuperação de arquivos, capaz de registrar e restaurar dados excluídos sem o auxílio de ferramentas externas, preservando a consistência estrutural e o controle de blocos em disco.

Neste trabalho foram implementadas as principais estruturas de um sistema de arquivos: o superbloco, o gerenciamento de blocos livres e a organização de metadados por meio de árvores B. A estrutura de *i-nodes*, por exemplo, adota um modelo híbrido de endereçamento, combinando ponteiros diretos e árvores de intervalos (`bspans`), acompanhado de um mecanismo de recuperação de arquivos excluídos mantido em uma árvore própria. Também foram desenvolvidos utilitários de formatação e inspeção, como `mkfs.bsfs` e o interpretador `browser`. Por fim, o trabalho inclui testes comparativos de desempenho entre o BSFS e sistemas amplamente utilizados no Linux (Ext4, Btrfs e ZFS), avaliando o impacto do modelo proposto no desempenho e na viabilidade de operação em espaço de usuário.

O desenvolvimento de um sistema de arquivos experimental oferece uma oportunidade para compreender, de forma prática e controlada, o funcionamento interno dos mecanismos de armazenamento de dados. Trabalhos dessa natureza permitem investigar a relação entre estruturas de metadados, estratégias de alocação e políticas de recuperação, aspectos que geralmente permanecem ocultos nas implementações complexas de sistemas de arquivos modernos.

1.2 Estrutura do Trabalho

Este trabalho está organizado em cinco capítulos, além desta introdução inicial. O Capítulo 2 apresenta os fundamentos teóricos necessários para compreender o funcionamento e a organização interna dos sistemas de arquivos. São discutidos seus objetivos, a estrutura física de uma partição, o papel do superbloco, gerenciamento de blocos livres, *i-nodes*, os métodos de alocação de blocos, a implementação de diretórios, e os compromissos entre desempenho e confiabilidade que orientam o projeto dessas estruturas.

O Capítulo 3 descreve sistemas de arquivos utilizados em ambientes Linux: Ext2, Ext3, Ext4, ZFS e Btrfs, com ênfase nas estratégias de gerenciamento e otimização da alocação de blocos, organização de metadados e métodos de preservação de integridade. Essa fundamentação estabelece o contexto necessário para a análise e o desenvolvimento da proposta apresentada neste trabalho.

O Capítulo 4 apresenta a implementação do *Basic Solution File System*. São detalhadas suas principais estruturas, incluindo o superbloco, o mapeamento de blocos livres, as árvores B utilizadas para indexação e gerenciamento de *i-nodes*, diretórios e blocos de dados, além do mecanismo de recuperação de arquivos excluídos. O capítulo também descreve os utilitários de formatação e interação com o sistema, bem como os resultados dos testes comparativos de desempenho.

Por fim, o Capítulo 5 apresenta as conclusões do trabalho, sintetizando os resultados obtidos, as limitações identificadas e as perspectivas de aprimoramento futuro, com destaque para a implementação do sistema em espaço de *kernel*, a implementação de métodos de verificação de integridade e novas estratégias de alocação e paralelismo.

Capítulo 2

Sistemas de Arquivos

> *We build our computer the way we build our cities: over time, without a plan, on top of ruins.*
>
> – Ellen Ullman

Neste capítulo, serão abordados o funcionamento e a organização interna dos sistemas de arquivos do ponto de vista teórico. Sistemas de arquivos tem, por definição, os seguintes objetivos: organizar, persistir e controlar o acesso aos dados em um sistema operacional. Embora seja transparente ao usuário na maior parte do tempo, são comparadas a visão do usuário, que interage com nomes e conteúdos de arquivos, com a perspectiva do sistema operacional, responsável por gerenciar as sequências de *bytes* e metadados por meio de chamadas de sistema.

Essa sequência de *bytes* e metadados é organizada em blocos dentro de um dispositivo. De forma geral, sistemas de arquivos estruturam espaço físico em áreas distintas dedicadas à descrição global do sistema, ao gerenciamento do espaço livre, ao armazenamento dos metadados dos arquivos e aos dados propriamente ditos. A forma como essas áreas são implementadas varia conforme o sistema de arquivos, mas todas têm por objetivo possibilitar o controle e a localização eficiente das informações gravadas em disco.

Em seguida, o capítulo apresenta o *i-node* como o elemento central de metadados e endereçamento de blocos, a partir do qual se exploram os métodos de alocação de espaço e suas implicações diretas, como o desempenho em leitura e a fragmentação. Adicionalmente, são detalhados o funcionamento dos diretórios, o gerenciamento do espaço livre e os critérios para a escolha do tamanho de bloco, encerrando com os objetivos que guiam projetos modernos de sistemas de arquivos, que busca equilibrar desempenho, confiabilidade e eficiência de armazenamento.

2.1 O que é um Sistema de Arquivos

Um sistema de arquivos é a estrutura responsável por organizar, de forma eficiente, as operações de leitura, gravação e acesso às informações armazenadas em dispositivos de bloco. Dispositivos de bloco são, normalmente, atribuídos a memórias mais lentas como, por exemplo, discos mecânicos (*hard drives*), discos de estado sólido (*solid state drive*) ou outros dispositivos de armazenamento não volátil. Dessa forma, um sistema de arquivos é uma abstração para a complexidade do armazenamento físico, oferecendo ao sistema operacional e aos usuários uma forma estruturada, segura e consistente de lidar com os dados. Além disso, também atua como intermediário entre o *hardware* e o *software*, garantindo que as informações sejam acessadas e gerenciadas independentemente da natureza do dispositivo de armazenamento.

Com o surgimento dos discos magnéticos, passou a ser possível acessar registros de arquivos fora da ordem em que foram gravados. Esse tipo de acesso, chamado de *acesso aleatório*, tornou o processo de leitura de dados mais rápido, já que não é mais necessário percorrer todos os registros anteriores para chegar a uma posição específica. Dessa forma, dois métodos são comuns: o `read`, que lê a partir de uma posição inicial definida; e o `seek`, que move o ponteiro para uma posição exata dentro do arquivo, permitindo a leitura de forma sequencial [17].

Ao analisarmos a organização de um sistema de arquivos, identificamos duas perspectivas fundamentais: a do usuário e a do sistema operacional. Do ponto de vista do usuário, um aspecto relevante é a forma como os arquivos são visualizados e manipulados, ou seja, o que constitui um arquivo, seu nome, tipo e conteúdo. O nome dos arquivos, por exemplo, é um mecanismo que permite aos usuários simplificar o processo de gerenciamento e organização dos seus dados. Por outro lado, o sistema operacional, normalmente utiliza informações vinculadas aos metadados dos cabeçalhos dos arquivos. Isso não é, necessariamente, verdade para todos os sistemas, mas é comum em derivados do Unix.

Sistemas operacionais modernos adotam o modelo de arquivos não estruturados, no qual qualquer tipo de dado pode ser armazenado sem imposições sobre sua organização interna. Entretanto, diferentes modelos de organização existem. No modelo de arquivo como sequência de *bytes*, o sistema enxerga o arquivo apenas como um fluxo linear de dados, cabendo à aplicação interpretar seu conteúdo. Já no modelo baseado em registros de tamanho fixo, o arquivo é dividido em unidades uniformes, facilitando acesso direto a posições específicas. Por fim, em estruturas organizadas em árvore, o arquivo é composto por registros hierárquicos contendo campos-chave, o que permite buscas otimizadas por meio de navegação estruturada [18].

Outra característica associada a um sistema de arquivos são as operações sobre os arquivos armazenados. Essas operações incluem criação, remoção, leitura, escrita, modificação de permissões e movimentação entre diretórios, todas executadas por chamadas ao sistema operacional, que garante acesso controlado e seguro aos dados.

Por exemplo, ao abrir um arquivo, o sistema carrega (total ou parcialmente) suas informações na memória para acessos subsequentes; ao fechar, garante que os dados pendentes sejam escritos corretamente no disco. Operações como leitura e escrita manipulam os dados com base na posição atual do ponteiro do arquivo, e algumas chamadas permitem navegar aleatoriamente pelo seu conteúdo [1].

2.2 A Estrutura do Sistema de Arquivos

Os sistemas de arquivos apresentam estruturas variadas, mas compartilham componentes e princípios de organização e funcionamento. Entre os elementos mais comuns estão o superbloco, responsável pelas informações de controle global, os blocos de inicialização e, em muitos sistemas, estruturas de mapeamento como os *bitmaps*, utilizadas para indicar quais blocos estão livres ou ocupados.

Durante o processo de inicialização do computador, a BIOS é responsável por localizar e executar o Registro Mestre de Inicialização (Master Boot Record — MBR). Esse registro contém o código que identifica a partição ativa do disco, lê seu primeiro bloco e transfere o controle da execução [18]. A partir desse ponto, o carregamento do sistema operacional segue conforme a estrutura específica da partição. Embora o princípio geral seja comum, a organização interna das partições varia entre diferentes sistemas de arquivos, que frequentemente incluem componentes ilustrados na Figura 2.1.

Nos sistemas atuais, o processo de inicialização é geralmente realizado por meio da UEFI (*Unified Extensible Firmware Interface*), que substitui a BIOS e o esquema tradicional de particionamento baseado em MBR. A UEFI utiliza a tabela GPT (GUID *Partition Table*), que permite um número maior de partições, tamanhos superiores a 2 TiB e mecanismos adicionais de integridade. Em vez de carregar um bloco fixo como o MBR, a UEFI localiza e executa um arquivo de *boot* armazenado em uma partição especial denominada EFI *System Partition* (ESP). Apesar dessas diferenças estruturais, o papel desempenhado pelo sistema de *firmware* permanece equivalente: preparar o ambiente inicial e transferir o controle para o carregador do sistema operacional, que então prossegue com a montagem e utilização do sistema de arquivos.

O superbloco constitui a estrutura central de controle do sistema de arquivos, responsável por armazenar seus principais parâmetros de configuração e integridade. É carregado na memória durante a inicialização do sistema ou no momento em que o sistema de arquivos é montado. Entre os dados mantidos nessa estrutura estão o *número mágico*, um identificador único que permite ao sistema operacional reconhecer o tipo de sistema de arquivos, e a quantidade total de blocos disponíveis.

Além disso, o superbloco armazena dados administrativos relevantes para o funcionamento do sistema de arquivos. Entre eles, podem ser citados: o tamanho total do sistema de arquivos e de cada bloco de armazenamento, a quantidade de blocos e *i-nodes* livres, a posição inicial e a extensão da tabela de *i-nodes*, e a localização

dos mapas de *bits* responsáveis pelo controle de blocos e *i-nodes*. Também podem ser registrados parâmetros de controle, como a data e a hora do último acesso ou montagem, o número de montagens e indicadores de integridade, que sinalizam se o sistema de arquivos foi desmontado de forma adequada. Essas informações permitem ao sistema operacional garantir a consistência da estrutura e realizar operações de alocação, acesso e manutenção.

Nos blocos seguintes da partição são armazenadas as informações referentes à alocação de blocos disponíveis no sistema de arquivos, geralmente representadas por meio de um *bitmap* ou, em algumas implementações, por listas encadeadas de ponteiros. Em sistemas inspirados no modelo Unix, essa região é seguida pelas estruturas de metadados conhecidas como *i-nodes*, responsáveis por descrever as propriedades e os endereços de dados de cada arquivo. Em outros sistemas de arquivos são utilizadas estruturas funcionais equivalentes, mas organizadas e nomeadas de forma diferente.

Em seguida, é encontrado o diretório raiz, representando o nível mais alto da árvore hierárquica do sistema de arquivos. Por fim, a área remanescente do disco é utilizada para armazenar os demais diretórios e arquivos pertencentes ao sistema.

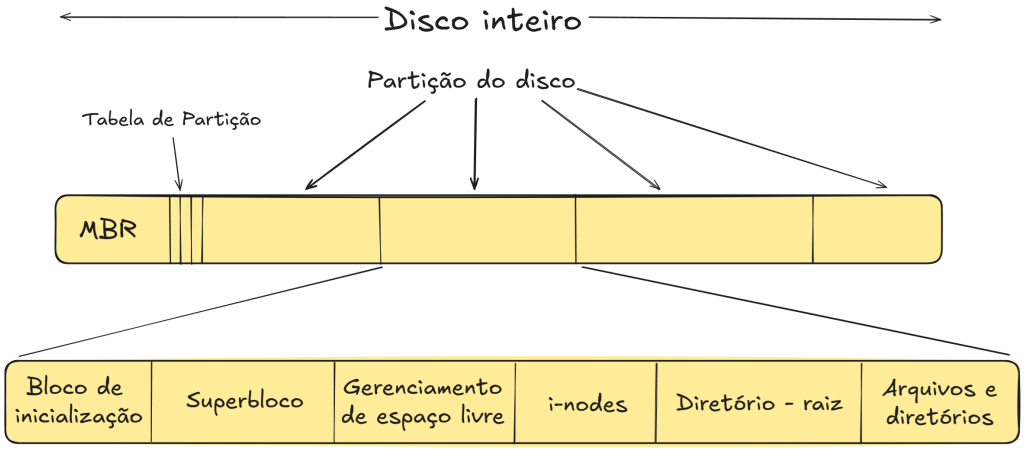


Figura 2.1: Estrutura simplificada de partição de disco e seus componentes [18].

2.2.1 Estrutura de Dados *i-node*

Uma estratégia empregada por sistemas de arquivos para monitorar a quais blocos de disco cada arquivo pertence consiste em associá-lo a uma estrutura de dados denominada *i-node* (*index-node* — nó-índice). Essa estrutura armazena metadados do arquivo e os endereços dos blocos onde seus dados residem. Com esse descritor, é possível localizar no disco os blocos que compõem o arquivo, conforme o esquema mostrado na Figura 2.2.

A principal vantagem do modelo baseado em *i-nodes* é que apenas os *i-nodes* associados a arquivos abertos precisam permanecer na memória principal. Assim, o

consumo de memória depende do número de arquivos abertos simultaneamente, e não da quantidade total de arquivos armazenados no sistema. Esse comportamento é diferente das abordagens baseadas em tabelas globais de alocação, cujo tamanho cresce proporcionalmente à capacidade do disco e aumenta linearmente à medida que o dispositivo se expande [17].

A estrutura de *i-nodes* possui a limitação de conter um número fixo de ponteiros para endereçar blocos de dados. Quando o arquivo ultrapassa esse limite, é necessário adotar um mecanismo de expansão. A solução tradicional é o uso de blocos de indireção, nos quais um dos ponteiros do *i-node* referencia um bloco intermediário que armazena novos endereços de dados. Esse método pode ser estendido a múltiplos níveis, permitindo ampliar a capacidade de endereçamento sem aumentar o tamanho da estrutura, conforme ilustrado na Figura 2.2.

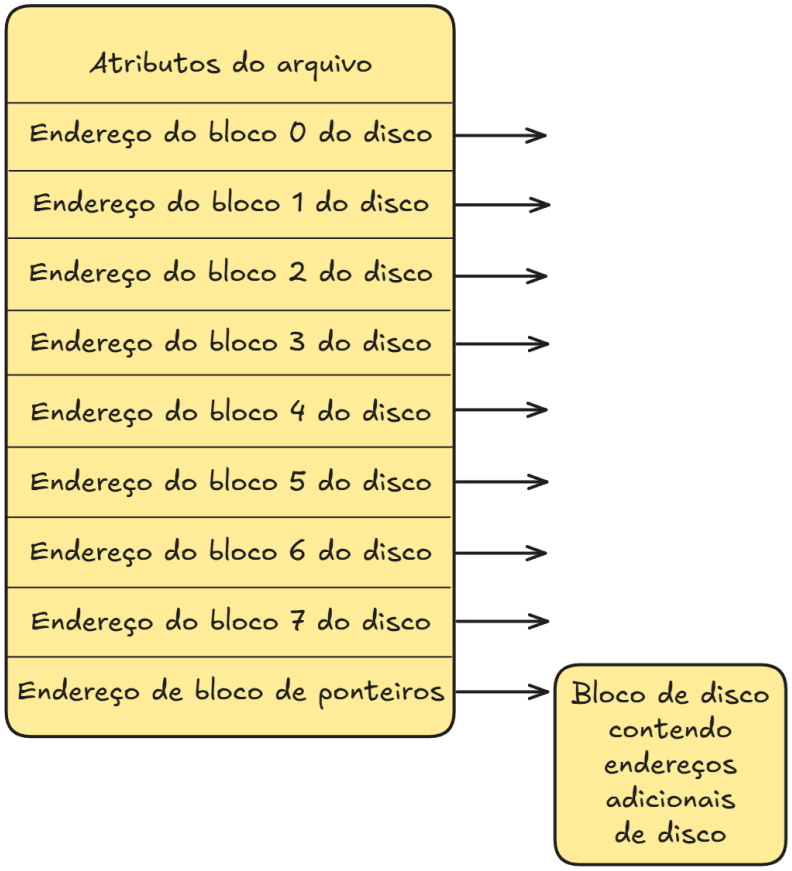


Figura 2.2: Exemplo de estrutura de *i-node* [18].

2.2.2 Armazenamento de dados contíguos

O armazenamento contíguo consiste na gravação de arquivos em blocos dispostos de forma sequencial no disco, de modo que todos os blocos de um mesmo arquivo ocupem posições adjacentes no dispositivo de armazenamento. Nesse modelo, um

arquivo de 50 KiB em um sistema com blocos de 1 KiB seria alocado em 50 blocos consecutivos. Por outro lado, com blocos de 2 KiB, ocuparia 25 blocos consecutivos [18].

A Figura 2.3 apresenta um exemplo de alocação em armazenamento contíguo. Os primeiros 18 blocos de disco são exibidos, iniciando pelo bloco 0 à esquerda. Inicialmente, o disco encontra-se vazio. Posteriormente, um arquivo denominado `prova.pdf`, com seis blocos de comprimento, é gravado a partir do bloco inicial. Em seguida, o arquivo `contas.txt`, de quatro blocos, é alocado imediatamente após o término de `prova.pdf`. E por fim, o `musica.mp3`, de oito blocos, é alocado imediatamente após o término do `contas.txt`.

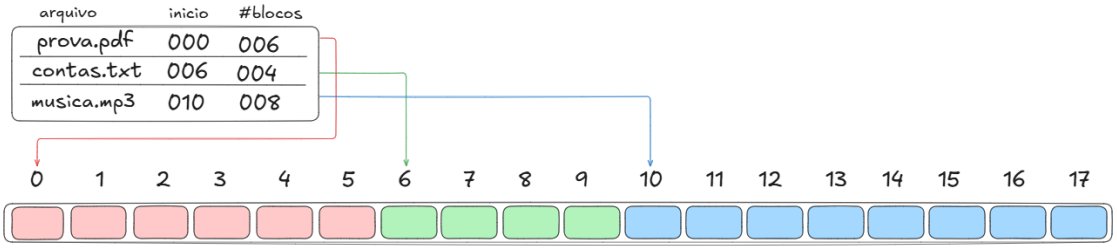


Figura 2.3: Distribuição contígua de blocos de disco para 3 arquivos.

Cada arquivo gravado inicia em um bloco completo. Dessa forma, quando o tamanho do arquivo não é múltiplo exato do tamanho do bloco, o espaço restante no último bloco permanece inutilizado, caracterizando uma fragmentação interna [18]. Um arquivo que ocupa 5,5 blocos utilizará efetivamente 6 blocos, sendo metade do último bloco espaço desperdiçado.

Essa forma de fragmentação ocorre porque o sistema de arquivos opera com unidades de alocação fixas, não sendo capaz de compartilhar o espaço livre de um bloco parcialmente ocupado entre diferentes arquivos. Embora a perda de espaço em arquivos isolados seja pequena, seu efeito cumulativo pode se tornar relevante em sistemas com grande número de arquivos, reduzindo a eficiência global de armazenamento.

A fragmentação interna difere da fragmentação externa, que surge quando o espaço livre está disperso em pequenas regiões não contíguas do disco, dificultando a alocação de novos blocos para arquivos grandes. Enquanto a fragmentação externa impacta o desempenho de leitura e escrita, a fragmentação interna afeta principalmente a utilização do espaço disponível.

Na alocação contígua, o controle de blocos de um arquivo é realizado por meio do endereço do primeiro bloco e da quantidade total de blocos que o compõem. Essa abordagem permite localizar qualquer bloco subsequente por meio de um cálculo baseado na posição inicial. Em operações de leitura, o acesso ao arquivo ocorre de forma contínua, uma vez que todos os blocos estão dispostos em sequência, o que

reduz o número de movimentações e as variações temporais associadas à busca por blocos dispersos no disco.

Entretanto, essa técnica apresenta uma limitação: a fragmentação externa do disco ao longo do tempo. Como ilustrado na Figura 2.4, quando um arquivo como `contas.txt` é removido, seus blocos são liberados, criando lacunas de espaço livre. O disco não é compactado imediatamente, uma vez que a operação de reorganização poderia exigir a movimentação de blocos de outros arquivos, tornando o processo mais custoso. Consequentemente, o disco passa a apresentar arquivos intercalados com lacunas de espaço livre.

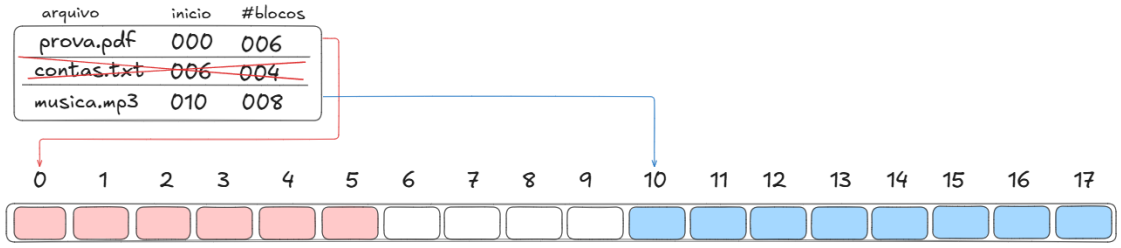


Figura 2.4: Distribuição contígua de blocos com 1 arquivo excluido.

Inicialmente, a fragmentação externa não representa um problema, pois novos arquivos podem ser gravados ao final do disco. Contudo, ao atingir a capacidade máxima, será necessário optar por compactação ou reutilização das lacunas disponíveis. Esta última abordagem requer a manutenção de uma lista de espaços livres e a necessidade de conhecer previamente o tamanho final do arquivo para escolher uma lacuna adequada. Este procedimento é pouco prático, pois exigiria que o usuário informasse o tamanho final de cada documento ao iniciar sua criação, o que pode resultar em falhas ou na necessidade de reiniciar o processo diversas vezes até localizar um espaço adequado.

A alocação contígua também apresenta limitações relacionadas à gestão de espaço e à recuperação de falhas. Como os dados são gravados em blocos sequenciais, um arquivo só pode ser alocado se houver um espaço contíguo suficientemente grande, o que se torna cada vez mais difícil à medida que a partição se fragmenta. Dessa forma, mesmo com blocos livres disponíveis, arquivos extensos podem não encontrar regiões contínuas adequadas, resultando em subutilização do disco.

Além disso, falhas durante a escrita, como interrupções de energia ou erros de sistema, podem deixar blocos parcialmente ocupados ou metadados inconsistentes. Esse problema não é exclusivo da alocação contígua, mas torna-se mais crítico em sistemas que não implementam mecanismos de recuperação, como o *journaling*, responsáveis por registrar operações pendentes e restaurar o estado consistente do sistema após uma falha.

Historicamente, a alocação contígua foi amplamente utilizada em sistemas de

arquivos baseados em discos magnéticos, devido à sua simplicidade e alto desempenho. Porém, sua exigência de definir o tamanho final do arquivo no momento da criação e a crescente fragmentação resultante de operações de escrita e exclusão sucessivas levaram ao desenvolvimento de métodos mais flexíveis, como a alocação encadeada e a indexada. Em mídias de gravação única, como CD-ROMs e DVDs, onde o conteúdo é definido previamente e não sofre modificações, a alocação contígua permaneceu uma solução prática e eficiente.

2.2.3 Gerenciamento de espaço livre

O gerenciamento do espaço livre é um dos aspectos fundamentais no projeto de um sistema de arquivos, pois determina como os blocos disponíveis em disco são identificados, reservados e reutilizados. Diferentes estratégias de alocação influenciam diretamente o desempenho, a fragmentação e a eficiência do uso do armazenamento, exigindo um equilíbrio entre simplicidade de implementação e aproveitamento do espaço.

Existem duas abordagens para o armazenamento de arquivos: a alocação de dados em regiões sequenciais do disco e a divisão do conteúdo em blocos de tamanho fixo, que podem ser distribuídos em diferentes áreas do dispositivo. Essa escolha é análoga às estratégias de gerenciamento de memória, nas quais se utilizam modelos baseados em segmentação ou paginação para organizar o espaço disponível.

O armazenamento de um arquivo como uma sequência de *bytes* apresenta a desvantagem de exigir movimentação física do arquivo no disco quando há necessidade de expansão. Essa operação envolve a realocação de dados em regiões distintas do disco, diferentemente da movimentação de segmentos na memória principal.

Em razão dessa limitação, a maioria dos sistemas de arquivos modernos adotam a estratégia de dividir os arquivos em blocos de tamanho fixo [18]. Essa abordagem elimina a necessidade de contiguidade, permitindo maior flexibilidade na alocação de espaço e reduzindo o impacto de fragmentações. Os blocos podem ser posicionados em diferentes partes do disco, sendo a localização de cada um registrada em estruturas específicas do sistema de arquivos, como os *i-nodes*.

A definição do tamanho dos blocos de armazenamento constitui uma etapa essencial no projeto de sistemas de arquivos, pois influencia diretamente o equilíbrio entre desempenho e eficiência de utilização do espaço.

Blocos de tamanho maior reduzem a sobrecarga de gerenciamento e permitem que cada operação de leitura ou escrita inclua uma quantidade maior de dados, favorecendo o desempenho em arquivos que ocupam múltiplos blocos. Essa configuração, entretanto, aumenta o desperdício de espaço em arquivos que utilizam apenas uma fração de um bloco, uma vez que cada arquivo deve ocupar ao menos um bloco completo, independentemente da quantidade efetiva de dados armazenados.

Blocos de tamanho reduzido, por sua vez, aproveitam melhor o espaço disponível,

mas exigem um número maior de operações de entrada e saída (I/O) para arquivos que ocupam múltiplos blocos, elevando o tempo de acesso e o custo de manutenção dos metadados. Dessa forma, a escolha do tamanho de bloco representa um compromisso entre eficiência de armazenamento e desempenho operacional, aplicável tanto a discos magnéticos quanto a unidades de estado sólido (*SSDs*).

A definição do tamanho de bloco pode ser orientada por análises estatísticas da distribuição de tamanhos de arquivos em sistemas reais. Um estudo conduzido na Universidade Vrije e em um servidor *web* comercial indicou que, em 2005, aproximadamente 59,13% dos arquivos possuíam até 4 KiB e 90,84% até 64 KiB, com tamanho médio de 2475 *bytes* [19].

Considerando blocos de 1 KiB, entre 30% e 50% dos arquivos podem ser armazenados em um único bloco, enquanto blocos de 4 KiB elevam essa proporção para 60% a 70%. Observou-se também que cerca de 93% do espaço total do disco é ocupado por 10% dos arquivos que contêm a maior quantidade de dados. Dessa forma, o desperdício de espaço decorrente de arquivos que ocupam poucos blocos torna-se insignificante, uma vez que a maior parte da capacidade de armazenamento é consumida por arquivos de maior extensão. Assim, aumentar o tamanho dos blocos para acomodar uma fração maior dos arquivos menores tende a produzir efeitos limitados sobre a eficiência global de utilização do espaço.

Os resultados dessa análise são apresentados na Tabela 2.1. Os conjuntos "UV 1984" e "UV 2005" correspondem à levantamentos realizados nos sistemas de arquivos da Universidade de Vrije em diferentes períodos, enquanto as colunas "Web" referem-se à análise de arquivos armazenados em um servidor Web comercial.

Tamanho	**UV 1984**	**UV 2005**	**Web**	**Tamanho**	**UV 1984**	**UV 2005**	**Web**
1	1,79	1,38	6,67	16 KiB	92,53	78,92	86,79
2	1,88	1,53	7,67	32 KiB	97,21	85,87	91,65
4	2,01	1,65	8,33	64 KiB	99,18	90,84	94,80
8	2,31	1,80	11,30	128 KiB	99,84	93,73	96,93
16	3,32	2,15	11,46	256 KiB	99,96	96,12	98,48
32	5,13	3,15	12,33	512 KiB	100,00	97,73	98,99
64	8,71	4,98	26,10	1 MiB	100,00	98,87	99,62
128	14,73	8,03	28,49	2 MiB	100,00	99,44	99,80
256	23,09	13,29	32,10	4 MiB	100,00	99,71	99,87
512	34,44	20,62	39,94	8 MiB	100,00	99,86	99,94
1 KiB	48,05	30,91	47,82	16 MiB	100,00	99,94	99,97
2 KiB	60,87	46,09	59,44	32 MiB	100,00	99,97	99,99
4 KiB	75,31	59,13	70,64	64 MiB	100,00	99,99	99,99
8 KiB	84,97	69,96	79,69	128 MiB	100,00	99,99	100,00

Tabela 2.1: Distribuição percentual do tamanho de arquivos em diferentes conjuntos de dados [18, 19].

Por outro lado, blocos menores aumentam o número de blocos por arquivo,

tornando a leitura mais lenta devido a buscas e atrasos rotacionais, exceto em discos de estado sólido. Considerando um disco rígido com 1 MiB por trilha, tempo de rotação de 8,33 ms e tempo de busca de 5 ms, o tempo de leitura de um bloco de k *bytes* é dado por:

$$t_{\text{leitura}} = 5 + 4,165 + \frac{k}{1.000.000} \times 8,33 \text{ ms}$$

A Figura 2.5 ilustra a taxa de transferência em função do tamanho do bloco. Para estimar a eficiência de espaço, assume-se que todos os arquivos possuam 4 KiB, valor próximo ao observado na Universidade Vrije. A curva sólida da figura mostra a eficiência de espaço, enquanto a curva tracejada indica a taxa de dados.

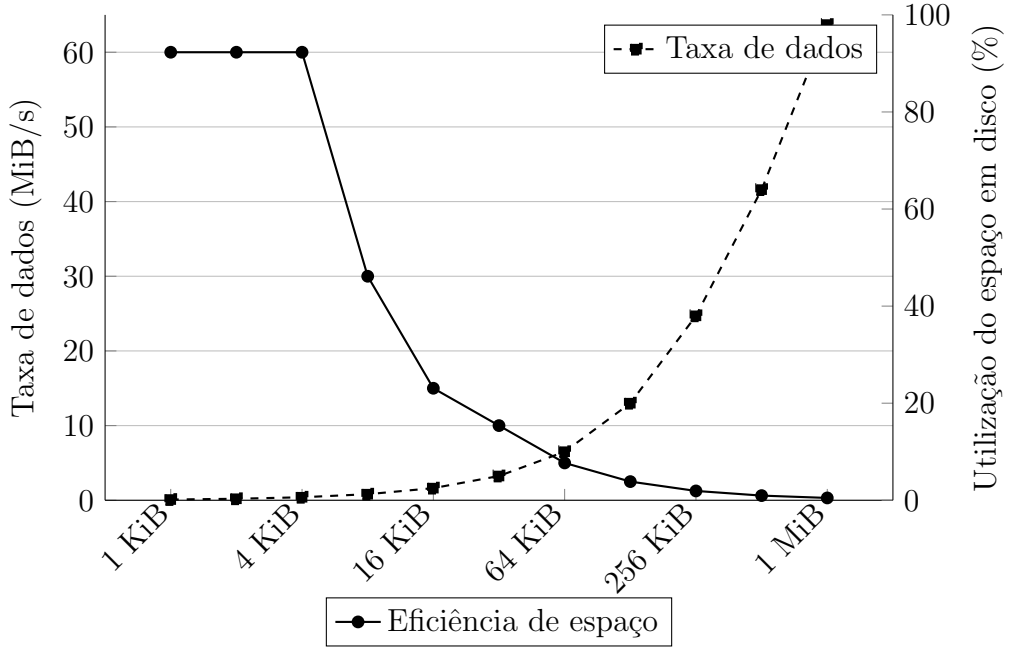


Figura 2.5: Taxa de dados e utilização de espaço em disco rígido em função do tamanho do bloco. [18].

O tempo de acesso é dominado pelo tempo de busca e atraso rotacional. Quanto maiores os blocos, mais eficiente é a transferência de dados, até que o tempo de transferência se torne relevante. Portanto, a escolha do tamanho de bloco deve equilibrar desperdício de espaço e desempenho de leitura.

A análise do tamanho de bloco não se limita ao desempenho, mas também deve considerar a eficiência de espaço. Suponha arquivos de 4 KiB sendo armazenados em blocos de 1 KiB, 2 KiB ou 4 KiB. Nesse caso, os arquivos ocupam 4, 2 e 1 bloco(s), respectivamente, sem desperdício de espaço. Entretanto, se o bloco for maior que o arquivo, como 8 KiB ou 16 KiB, a eficiência de espaço cai para 50% e 25%, respectivamente. Na prática, poucos arquivos possuem tamanho múltiplo

exato do bloco, de modo que sempre haverá algum desperdício no último bloco de cada arquivo.

A análise dos dados evidencia um conflito entre desempenho e eficiência na utilização do espaço de armazenamento. Blocos de menor tamanho favorecem o aproveitamento do disco, mas aumentam o número de acessos necessários para manipular arquivos extensos, reduzindo o desempenho. Em contrapartida, blocos maiores elevam a taxa de transferência, porém resultam em desperdício de espaço. No conjunto de dados analisado, não há um ponto de equilíbrio ideal: o tamanho de bloco que mais se aproxima do cruzamento entre as curvas de desempenho e eficiência é de 64 KiB, apresentando uma taxa de 6,6 MiB/s e eficiência de 7%, valores insatisfatórios para ambos os critérios.

Historicamente, sistemas de arquivos têm adotado tamanhos de bloco entre 1 KiB e 4 KiB; contudo, em dispositivos atuais com capacidades superiores a 1 TiB, blocos de 64 KiB podem se mostrar viáveis, mesmo com certo desperdício de espaço, devido à ampla disponibilidade de armazenamento nos dispositivos.

Estudos empíricos corroboram essa relação entre o tamanho de bloco e o padrão de uso dos arquivos. Vogels [22] realizou uma análise do comportamento de sistemas de arquivos no ambiente Windows NT, comparando-o a sistemas baseados em UNIX. O estudo revelou que o modelo de gerenciamento de arquivos do Windows NT é mais complexo, envolvendo um número maior de chamadas de sistema mesmo para operações simples de edição, o que aumenta a sobrecarga de entrada e saída (I/O) no sistema.

Apesar das diferenças de implementação, os resultados observados no Windows NT apresentaram tamanhos médios ponderados de arquivos semelhantes aos registrados em sistemas UNIX. Arquivos acessados apenas para leitura possuíam tamanho médio de aproximadamente 1 KiB, enquanto arquivos apenas escritos apresentavam cerca de 2,3 KiB, e aqueles lidos e escritos atingiam, em média, 4,2 KiB. Esses valores demonstram consistência com os resultados obtidos na Universidade Vrije, indicando que a maioria dos arquivos ocupa poucos blocos, o que reforça a relevância da escolha criteriosa do tamanho de bloco para equilibrar eficiência e desempenho.

2.2.4 Gerenciamento de blocos livres

Após a definição do tamanho de bloco, torna-se necessário estabelecer um método para o controle dos blocos livres no disco. O gerenciamento dessa informação é utilizado para a alocação de novos arquivos e para o reaproveitamento do espaço liberado. Entre as técnicas mais empregadas, destacam-se a lista encadeada de blocos livres e o mapa de *bits* (*bitmap*), ilustradas na Figura 2.6.

Na abordagem baseada em lista encadeada, cada bloco da lista armazena uma sequência de endereços de blocos livres, além de um ponteiro para o próximo elemento da lista. O número de endereços possíveis em cada bloco depende de seu tamanho e da largura dos identificadores de bloco. Por exemplo, em um sistema

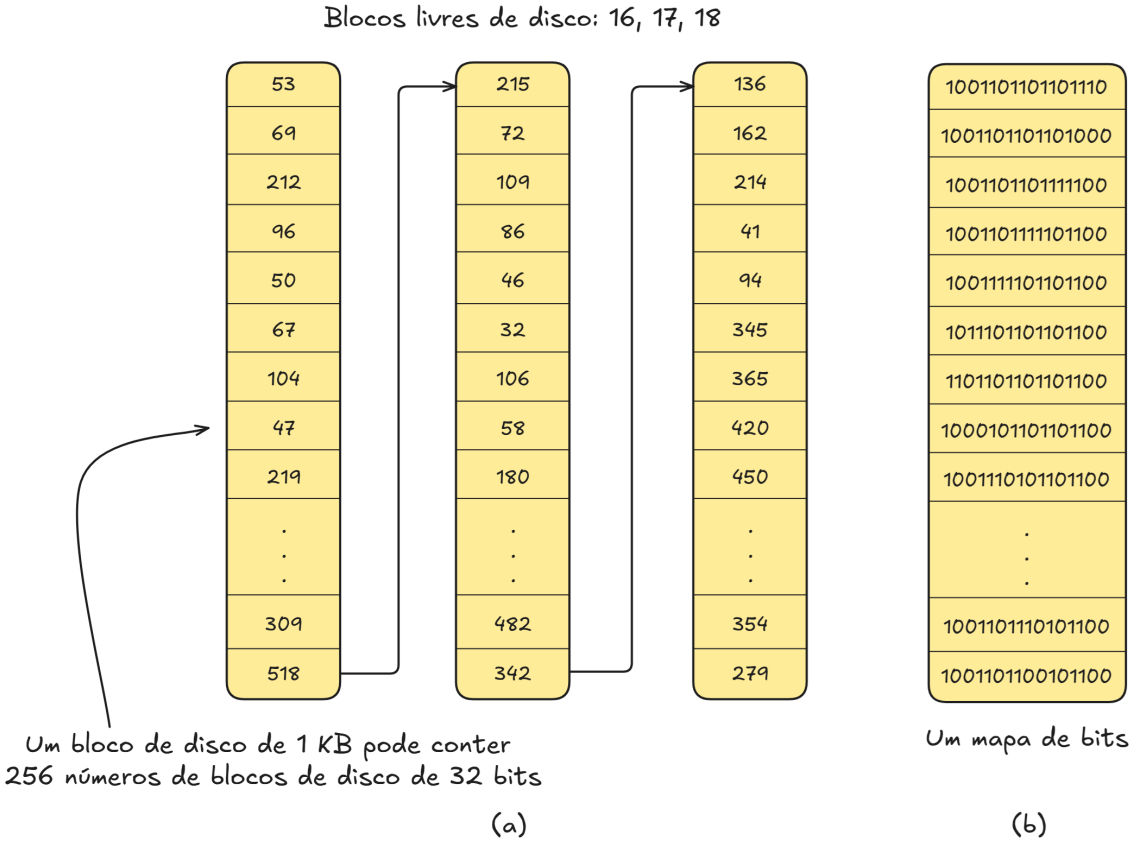


Figura 2.6: (a) Lista encadeada utilizada para armazenar os blocos livres; (b) Mapa de *bits* (*bitmap*). [18].

com blocos de 1 KiB e endereços de 32 *bits*, cada bloco pode conter até 255 referências a blocos livres, reservando uma posição para o ponteiro que encadeia o próximo bloco da lista. Em um disco de 1 TiB, composto por aproximadamente 1 bilhão de blocos, essa estrutura exigiria cerca de 4 milhões de blocos para armazenar a lista completa.

A segunda técnica de controle é o *bitmap*, que representa o estado de ocupação de cada bloco do dispositivo por meio de uma sequência compacta de *bits*. Cada posição do mapa indica se o respectivo bloco está livre ou alocado, permitindo que a verificação e a atualização dessas informações sejam realizadas de forma direta.

Em um disco de 1 TiB, o *bitmap* requer aproximadamente 130.000 blocos de 1 KiB para seu armazenamento, o que o torna mais econômico que a lista encadeada, já que utiliza uma quantidade fixa e reduzida de *bits* para representar cada bloco. Essa abordagem apresenta melhor desempenho em cenários de alta utilização do disco, embora, em situações de espaço quase esgotado, a lista encadeada possa oferecer vantagem por lidar com um número menor de referências livres.

Quando os blocos livres tendem a ocorrer em sequências de blocos consecutivos,

a lista encadeada de blocos livres pode ser otimizada para gerenciar conjuntos de blocos em vez de blocos individuais. Um contador pode ser associado a cada bloco, indicando a quantidade de blocos livres consecutivos. No melhor cenário, um disco formatado ou com baixa fragmentação poderia ser representado por apenas dois números: o endereço do primeiro bloco livre e o contador de blocos consecutivos.

Em contrapartida, se o disco estiver fragmentado, o controle de conjuntos torna-se menos eficiente, pois além do endereço, é necessário armazenar também o contador. Essa técnica ilustra um dos dilemas recorrentes no projeto de sistemas de arquivos: o equilíbrio entre simplicidade estrutural e eficiência operacional depende do padrão de utilização do armazenamento.

No método baseado em lista encadeada, apenas um bloco de ponteiros é mantido na memória principal, servindo como referência temporária para as operações de alocação e liberação de blocos. Durante a criação de arquivos, os endereços são obtidos desse bloco; quando esgotado, um novo é lido do disco. De modo análogo, na remoção de arquivos, os blocos liberados são adicionados a essa estrutura e, ao atingir sua capacidade, o bloco de ponteiros é escrito de volta ao disco. Esse mecanismo pode gerar sobrecarga de entrada e saída (I/O) em cenários com arquivos pequenos e temporários, devido à frequência com que blocos precisam ser lidos e gravados novamente, aumentando o número de operações.

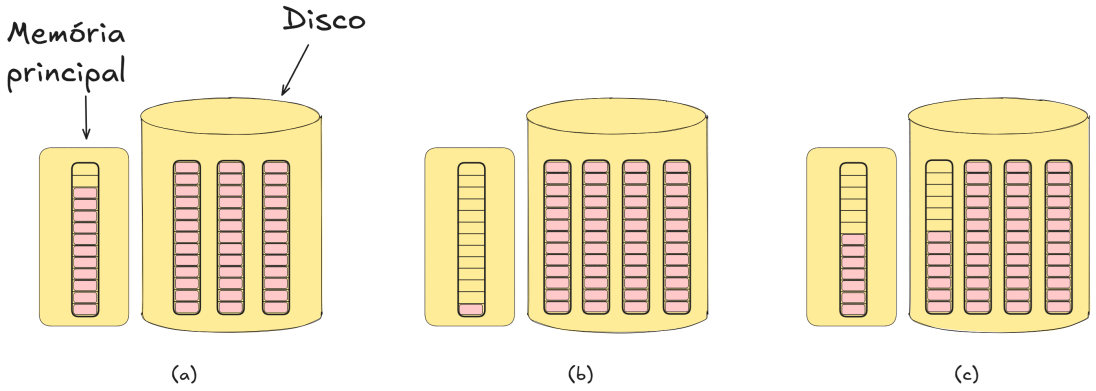


Figura 2.7: (a) Bloco na memória principal quase completo, contendo ponteiros para blocos livres no disco, enquanto três blocos de ponteiros permanecem armazenados no disco. (b) Situação após a remoção de um arquivo composto por três blocos. (c) Estratégia alternativa para tratar a liberação desses três blocos livres. As áreas sombreadas indicam ponteiros para blocos de disco disponíveis. [18].

A Figura 2.7 ilustra a diferença entre o comportamento tradicional da lista encadeada de blocos livres e uma estratégia alternativa que reduz operações de entrada e saída. No método convencional, mostrado nas partes (a) e (b), todo bloco liberado precisa ser imediatamente registrado no bloco de ponteiros mantido na memória principal; quando esse bloco fica cheio, ocorre um “transbordo”, exigindo que um novo bloco seja lido ou escrito em disco. A alternativa, apresentada na parte (c),

divide o bloco de ponteiros sempre que ele é preenchido, mantendo na memória apenas um bloco parcialmente cheio. Com isso, o sistema consegue realizar diversas operações de criação e remoção de arquivos temporários sem operar sobre o disco. A maior parte dos blocos de ponteiros permanece cheia e armazenada em disco, enquanto o bloco na memória continua parcialmente cheio, permitindo adicionar ou remover ponteiros sem incorrer imediatamente em operações de entrada e saída.

No método baseado em *bitmap*, apenas um bloco da estrutura é mantido na memória principal, sendo necessária a escrita em disco apenas quando ocorre seu preenchimento ou esvaziamento completo. Essa abordagem facilita a alocação de blocos contíguos, reduzindo o deslocamento físico da cabeça de leitura em dispositivos de armazenamento magnético. Além disso, por possuir tamanho fixo e previsível, o *bitmap* pode ser mapeado na memória virtual, permitindo a paginação de suas seções conforme a demanda e otimizando o uso dos recursos do núcleo do sistema operacional.

2.2.5 Alocação de blocos

A técnica de alocação define a forma como os blocos de disco são selecionados para armazenar os dados de um arquivo, influenciando o desempenho e a eficiência do sistema. Assim como no gerenciamento de memória principal, os algoritmos de alocação buscam localizar regiões livres de forma rápida e reduzir a fragmentação do espaço, equilibrando custo computacional e aproveitamento do armazenamento [18].

Entre as estratégias aplicáveis ao contexto de sistemas de arquivos estão os métodos *first fit*, *next fit*, *best fit*, *worst fit* e *quick-fit*. Nesses esquemas, os blocos de disco assumem o papel dos segmentos de memória, e as informações de disponibilidade são mantidas em estruturas auxiliares, como *bitmaps* ou listas encadeadas de blocos livres, permitindo o controle das áreas de armazenamento.

Entre os métodos utilizados, o *first fit* (primeiro encaixe) consiste em percorrer a estrutura de blocos livres até encontrar o primeiro intervalo contíguo de tamanho suficiente para atender à solicitação. No contexto de sistemas de arquivos, essa abordagem é simples e eficiente, pois reduz o tempo de busca e tende a preservar blocos contíguos nas áreas iniciais do disco. Entretanto, à medida que o espaço livre se fragmenta, o número de varreduras necessárias para localizar blocos disponíveis aumenta, impactando diretamente o tempo de escrita e a taxa de transferência.

Uma variação do método é o *next fit* (próximo encaixe), que armazena a posição da última alocação bem-sucedida e retoma a busca a partir desse ponto nas próximas solicitações. Essa estratégia busca evitar repetidas varreduras das mesmas regiões da partição, mas em sistemas de arquivos tende a apresentar desempenho semelhante ou ligeiramente inferior ao *first fit*, pois distribui a fragmentação ao longo de todo o espaço de armazenamento.

O algoritmo *best fit* (melhor encaixe) procura entre todos os blocos livres aquele cujo tamanho mais se aproxima da quantidade solicitada, tentando minimizar o

desperdício interno de espaço. Apesar da eficiência, esse método costuma resultar em fragmentação mais severa, pois tende a gerar pequenos blocos residuais que não são reutilizáveis.

O método *worst fit* (pior encaixe) segue a lógica inversa do *best fit*, escolhendo sempre o maior bloco livre disponível. A ideia é que, ao dividir uma grande região, o espaço remanescente ainda seja utilizável em futuras alocações. Na prática, entretanto, esse método não apresenta vantagens significativas em sistemas de arquivos e tende a gerar padrões de fragmentação semelhantes ao *best fit*.

Outra abordagem, o *quick fit*, mantém listas separadas de blocos livres para tamanhos pré-definidos. Em sistemas de arquivos, um mecanismo similar pode ser aplicado por meio de múltiplos mapas de *bits* ou tabelas de alocação que segmentam o espaço de acordo com a granularidade dos arquivos. Essa técnica reduz o tempo de busca por blocos, mas aumenta a complexidade de união e atualização dessas estruturas, especialmente quando arquivos são removidos ou redimensionados.

De modo geral, sistemas de arquivos modernos tendem a empregar variações do *first fit* combinadas a estratégias de agrupamento de blocos, como os *extents* abordados no Capítulo 3, para equilibrar simplicidade de implementação, desempenho e aproveitamento do espaço. Em implementações baseadas em *bitmaps*, a eficiência da alocação depende tanto do algoritmo de busca adotado quanto da forma como as regiões livres são representadas e atualizadas no disco.

2.2.6 Diretórios

Um diretório é uma estrutura de dados utilizada pelos sistemas de arquivos para organizar e manter a relação entre os nomes atribuídos aos arquivos e suas respectivas localizações físicas no disco. Funciona como um índice que associa identificadores legíveis pelo usuário a endereços ou descritores internos do sistema, permitindo o acesso hierárquico e estruturado aos dados armazenados. A organização em diretórios e subdiretórios compõe a árvore hierárquica do sistema de arquivos, na qual cada nó representa um agrupamento lógico de arquivos ou de outros diretórios.

No momento da abertura de um arquivo, o sistema operacional utiliza o caminho informado pelo usuário para localizar a entrada correspondente dentro do diretório. Essa entrada contém as informações necessárias para identificar os blocos de dados do arquivo. Dependendo do método de alocação adotado, essas informações podem incluir o endereço físico do arquivo (em sistemas de alocação contígua), o número do primeiro bloco (em listas encadeadas) ou o número do *i-node*, nos sistemas que utilizam essa estrutura. Em todos os casos, o diretório tem como função principal associar o nome do arquivo, armazenado em formato ASCII, às informações que permitem ao sistema localizar e manipular os dados de forma eficiente.

Além de armazenar referências para a localização dos dados, os diretórios também precisam registrar os atributos associados a cada arquivo. Esses atributos, como identificador do proprietário, permissões de acesso, datas de criação e modificação,

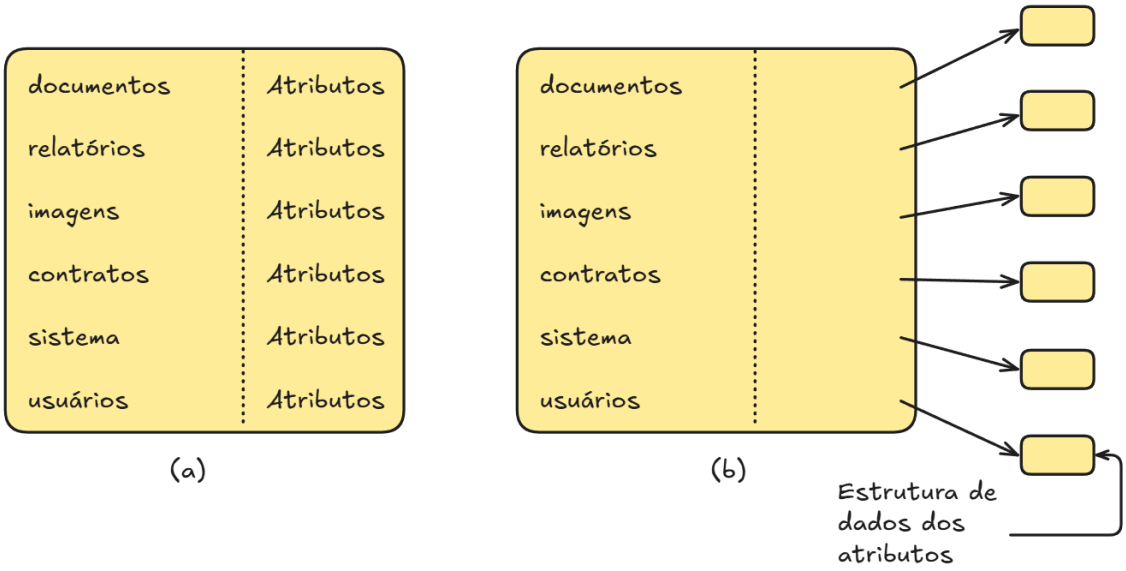


Figura 2.8: (a) diretório com entradas fixas que armazenam tanto os endereços de armazenamento quanto os atributos diretamente. (b) diretório em que cada entrada aponta apenas para um *i-node*, responsável pelos atributos e endereços de armazenamento [18].

são informações utilizadas para o controle e a segurança do sistema de arquivos. Uma das formas de organização consiste em armazenar esses atributos diretamente nas entradas do diretório. Nesse modelo, o diretório é composto por uma lista de registros de tamanho fixo, cada um correspondendo a um arquivo. Cada registro contém o nome do arquivo (também de tamanho fixo), a estrutura de atributos e um ou mais endereços de disco que indicam a localização dos blocos que compõem o conteúdo do arquivo.

Nos sistemas que utilizam *i-nodes*, os atributos dos arquivos, como permissões, proprietário, tamanho e marcações de tempo, são armazenados diretamente nos próprios *i-nodes*, e não nas entradas de diretório. Nessas implementações, cada entrada do diretório contém o nome do arquivo e o número de *i-node* correspondente, reduzindo o tamanho das entradas e tornando a estrutura mais eficiente em termos de armazenamento e acesso.

Historicamente, diferentes sistemas de arquivos adotaram restrições quanto ao tamanho e formato dos nomes de arquivos. O MS-DOS, por exemplo, utilizava o padrão 8.3, com nomes de até oito caracteres e uma extensão de três; já a Versão 7 do UNIX limitava o nome completo a 14 caracteres. Com o tempo, a demanda por maior expressividade e compatibilidade levou à adoção de nomes longos e de tamanho variável, comuns nos sistemas modernos. Essa flexibilidade, contudo, impõe desafios adicionais de implementação, uma vez que as entradas de diretório deixam de ter tamanho fixo e precisam acomodar nomes de diferentes comprimentos de forma eficiente.

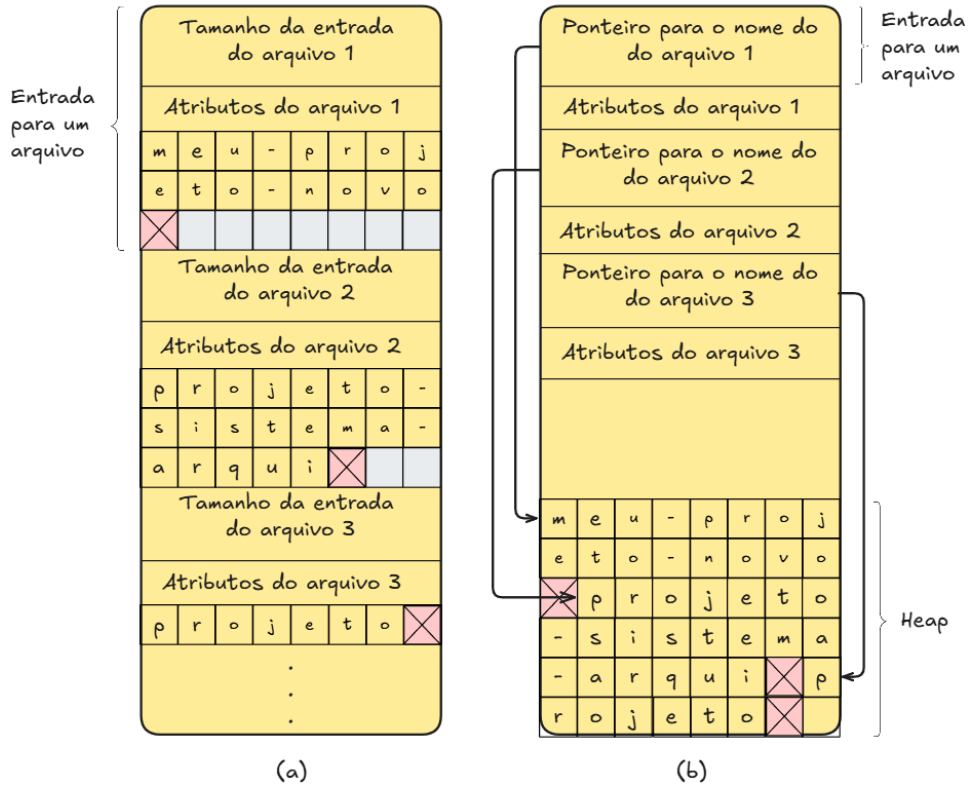


Figura 2.9: Duas abordagens para o gerenciamento de nomes de arquivos extensos em diretórios: (a) organização sequencial; (b) armazenamento em estrutura *heap*. [18]

Uma forma de implementação é definir um limite máximo para o tamanho dos nomes de arquivos e adotar um dos modelos apresentados na Figura 2.8, reservando, por exemplo, 255 caracteres para cada nome. Essa estratégia, apesar de funcional, resulta em desperdício de espaço, pois a maioria dos arquivos não utiliza nomes tão extensos.

Outra possibilidade consiste em não impor que todas as entradas de diretório possuam o mesmo tamanho. Nesse modelo, cada entrada contém uma parte fixa, que inicia com a informação sobre o tamanho da entrada, seguida de dados em formato definido, como o identificador do proprietário, o momento de criação, as permissões de acesso e outros atributos. Após essa estrutura de comprimento fixo, registra-se o nome do arquivo, independentemente do seu tamanho. Esse método está ilustrado na Figura 2.9(a), em um formato de organização em que o *byte* mais significativo aparece primeiro (*big-endian*).

No exemplo apresentado, há três arquivos: `project-budget`, `personnel` e `foo`. Cada nome de arquivo é encerrado com um caractere especial (geralmente o valor 0), representado na figura por um quadrado com a letra X. Para que cada entrada de diretório inicie em um limite correspondente ao tamanho de uma palavra, o nome de

cada arquivo é completado com caracteres adicionais até atingir esse alinhamento, o que é indicado pelas áreas cinzas da figura.

A desvantagem do método de armazenamento com nomes de tamanhos variáveis é que, ao remover um arquivo, cria-se uma lacuna de tamanho também variável no diretório. O próximo arquivo a ser inserido pode não se ajustar nesse espaço, ocasionando fragmentação interna. Esse problema é semelhante ao observado nos arquivos contíguos em disco. No entanto, como os diretórios são mantidos inteiramente na memória, torna-se possível compactá-los quando necessário, eliminando as lacunas formadas.

Outro ponto de atenção é que uma única entrada de diretório pode se estender por múltiplas páginas de memória, o que pode gerar falhas de página (*page faults*) durante a leitura de nomes de arquivos. Uma alternativa para evitar esse tipo de fragmentação consiste em adotar tamanhos fixos para as entradas de diretório e armazenar os nomes dos arquivos em uma área separada, denominada *heap*, localizada ao final de cada diretório, conforme ilustrado na Figura 2.9(b).

Esse método apresenta a vantagem de permitir que, ao remover uma entrada, o próximo arquivo inserido sempre encontre espaço disponível. Contudo, o *heap* precisa ser devidamente gerenciado, e falhas de página ainda podem ocorrer durante o processamento dos nomes. Um benefício adicional é que não há mais a necessidade de alinhar os nomes dos arquivos aos limites de palavra, eliminando a obrigatoriedade de preenchê-los com caracteres adicionais, como ocorria nas estruturas apresentadas anteriormente.

Nos modelos discutidos anteriormente, os diretórios são pesquisados linearmente, ou seja, o sistema percorre as entradas do início ao fim até encontrar o nome do arquivo desejado. Essa abordagem é simples e eficiente para diretórios pequenos, mas pode se tornar lenta em estruturas muito extensas. Para otimizar o desempenho em diretórios grandes, uma solução possível é o uso de tabelas de espalhamento (*hash tables*).

Nesse modelo, o diretório é implementado como uma tabela de espalhamento (*hash table*) composta por n posições, cada uma correspondendo a um possível valor de *hash*. O nome de cada arquivo é processado por uma função de espalhamento, que converte a sequência de caracteres em um número inteiro no intervalo de 0 a $n - 1$. Esse número determina a posição da tabela onde a entrada do arquivo será registrada. Quando múltiplos nomes produzem o mesmo valor de *hash*, ocorre uma colisão, resolvida pela criação de uma lista encadeada de entradas associadas àquela posição. Dessa forma, todas as entradas que compartilham o mesmo valor de *hash* permanecem acessíveis a partir de um único ponto da tabela, preservando a eficiência da busca.

Durante a busca, o sistema aplica a mesma função de espalhamento ao nome desejado para determinar qual lista encadeada consultar. Caso o nome não seja encontrado nessa lista, conclui-se que o arquivo não está presente no diretório. Essa

estratégia reduz o tempo de busca, embora aumente a complexidade da estrutura. Por essa razão, o uso de tabelas de espalhamento costuma ser mais vantajoso em sistemas em que os diretórios contêm centenas ou milhares de arquivos.

Entre as funções de espalhamento conhecidas, destaca-se o algoritmo *djb2*, proposto por Daniel J. Bernstein [2]. Trata-se de uma função de *hash* simples, frequentemente utilizada em sistemas que demandam operações rápidas de indexação. Seu funcionamento baseia-se em um valor inicial constante (geralmente 5381), que é atualizado iterativamente para cada caractere da cadeia de entrada por meio da expressão $hash = hash \times 33 + c$, em que c representa o código numérico do caractere processado. Essa operação produz uma boa distribuição de valores e reduz a ocorrência de colisões em conjuntos de dados textuais.

Outra alternativa para melhorar o desempenho em diretórios muito extensos consiste no uso de uma *cache* de buscas. Antes de iniciar uma pesquisa completa, o sistema verifica se o nome do arquivo já está armazenado na *cache*. Caso positivo, o arquivo pode ser localizado imediatamente. Esse método é eficiente quando a maioria das buscas recai sobre um conjunto relativamente pequeno de arquivos, permitindo um acesso quase instantâneo aos diretórios mais consultados.

2.3 Sumário

No presente capítulo foram apresentados os fundamentos dos sistemas de arquivos, relacionando a visão do usuário, centrada em nomes e conteúdos, com a perspectiva do sistema operacional, que manipula sequências de *bytes* e metadados por meio de chamadas de sistema. A explicação inicial destacou o papel do sistema de arquivos como camada de abstração entre *software* e *hardware*, responsável por garantir organização, persistência e acesso seguro aos dados.

Em seguida, foi detalhada a organização de uma partição de disco: o bloco de inicialização, o superbloco, as estruturas de mapeamento de espaço, com destaque para o uso de *bitmaps* a área de *i-nodes*, o diretório raiz e a região de dados. O *i-node* foi discutido como elemento central de metadados e endereçamento, e seus níveis de indireção para ampliar a capacidade de armazenamento sem aumentar o tamanho das estruturas mantidas em memória.

Em seguida, foram analisados os principais métodos de alocação e seus efeitos no desempenho e na fragmentação. A alocação contígua foi discutida quanto às vantagens em leitura sequencial e às limitações impostas pela fragmentação. Também foram avaliados os critérios para escolha do tamanho de bloco e as estratégias de gerenciamento de espaço livre, incluindo listas encadeadas e *bitmaps*.

Por fim, foram abordadas as estruturas de diretórios, comparando modelos de entradas fixas e variáveis, mecanismos de gerenciamento de nomes longos baseados em áreas *heap*, e técnicas de otimização de busca em diretórios extensos, como o uso de funções de espalhamento (*hashing*) e de *caches* de diretórios.

Capítulo 3

Trabalhos Relacionados

Trees sprout up just about everywhere in computer science.

– Donald E. Knuth

Este capítulo apresenta a evolução de sistemas de arquivos utilizados em ambientes Unix-*like*, com foco em características estruturais que influenciam desempenho, escalabilidade e integridade dos dados. A partir das primeiras implementações no Unix original e de propostas como o *Fast File System* (FFS), são abordados conceitos como *i-nodes*, diretórios hierárquicos, grupos de blocos e estratégias de gerenciamento de espaço livre, que servem de base para os sistemas adotados posteriormente para uso com o *kernel* Linux.

Na sequência, são analisados os sistemas da família *Extended File System* (Ext2, Ext3 e Ext4), que constituem a linha evolutiva dos sistemas de arquivos utilizados em distribuições Linux. Esses sistemas foram selecionados por representarem diferentes estágios de maturidade do modelo baseado em *i-nodes*, blocos de dados e grupos de blocos, oferecendo um panorama progressivo de soluções adotadas para problemas como fragmentação, escalabilidade de metadados, políticas de alocação e mecanismos de recuperação após falhas.

O capítulo também examina o ZFS e o Btrfs, dois sistemas de arquivos que implementam mecanismos avançados como *Copy-on-Write*, verificação de integridade por *checksums* e organização dinâmica de metadados por meio de árvores, tendências observadas em sistemas contemporâneos. Esses sistemas foram incluídos por incorporarem técnicas e soluções que serviram de base para o projeto do *Basic Solution File System*, especialmente em relação ao endereçamento por intervalos e utilização de estruturas em árvore para gerenciamento de metadados.

3.1 Sistemas de Arquivos UNIX

A evolução dos sistemas de arquivos utilizados com sistemas Unix acompanha o desenvolvimento das tecnologias de armazenamento, a partir do surgimento das primeiras versões do sistema nos laboratórios da AT&T em 1961. Como discutido no Capítulo 2, conceitos como *i-nodes*, diretórios hierárquicos, blocos de dados e mecanismos de gerenciamento de espaço livre tornaram-se elementos centrais no projeto de sistemas de arquivos. Entretanto, estes princípios foram consolidados de forma gradual, a partir das limitações observadas nas primeiras versões do Unix e das adaptações propostas ao longo das décadas seguintes [13].

O sistema de arquivos original do Unix, descrito por Ritchie [14], implementou mecanismos de organização que incluem a separação entre nomes e metadados por meio de *i-nodes* e o uso de diretórios armazenados como arquivos especiais. Embora suficiente para os dispositivos e cargas de trabalho da época, esse modelo apresentava limitações relacionadas à escalabilidade, ao dimensionamento dos blocos e à eficiência do gerenciamento de espaço.

Em resposta a essas restrições, diferentes variantes do Unix passaram a desenvolver sistemas de arquivos modificados. O *Berkeley Fast File System* (FFS), por exemplo, implementou o conceito de grupos de blocos, reorganizando *i-nodes* e dados de forma a reduzir movimentações físicas da cabeça de leitura em discos rígidos [11].

A partir dessas implementações, estratégias de gerenciamento de espaço livre também foram aprimoradas. Estruturas como listas encadeadas de blocos, *bitmaps* e contagem de sequências, descritas no Capítulo 2, passaram a ser aplicadas de maneira mais eficiente, reduzindo o custo de varreduras e melhorando a previsibilidade da alocação. Ao mesmo tempo, o aumento da variedade de tamanhos de arquivos exigiu modelos de alocação mais flexíveis que a alocação estritamente contígua.

O crescimento das árvores de diretórios em ambientes multiusuário também demandou abordagens otimizadas para nomeação, organização e busca. Entre as adaptações implementadas estão o suporte a nomes maiores, a adoção de entradas de diretório de tamanho variável e o uso de estruturas auxiliares para otimizar buscas, como tabelas de espalhamento e mecanismos de cache [11, 13]. Esses recursos complementaram a evolução dos sistemas de arquivos, aproximando-os das necessidades observadas em sistemas Unix modernos.

O *Minix File System* (MFS), utilizado no sistema operacional educacional Minix, adotava uma implementação simples, com tabelas fixas de *i-nodes* e blocos pequenos. Embora limitado em capacidade, seu desenho serviu como referência didática importante na compreensão das estruturas clássicas de sistemas de arquivos baseados em *i-nodes*.

Com a consolidação do *kernel* Linux nos anos 1990, tornou-se necessário um sistema de arquivos compatível com os modelos Unix, mas que incorporasse melhorias estruturais para suportar discos de maior capacidade e cargas de trabalho

mais intensas. O sistema Ext surgiu nesse contexto, derivado de soluções anteriores e inspirado tanto no *Minix File System* quanto no FFS. Porém, seu desempenho e suas limitações operacionais mostraram a necessidade de uma reformulação.

3.2 Ext2

O *Second Extended File System* (Ext2) foi apresentado em 1994 como sucessor do sistema Ext, consolidando-se como o primeiro sistema de arquivos amplamente adotado pelas distribuições Linux. Sua concepção foi influenciada pelo *Minix File System* e pelo *Fast File System* (FFS), a partir dos quais herdou o modelo baseado em *i-nodes* e blocos de dados. O Ext2 implementou avanços em relação ao seu antecessor ao incorporar mecanismos de gerenciamento de metadados mais eficientes e implementar grupos de blocos, projetado para reduzir a fragmentação externa e otimizar o acesso em discos rígidos [13].

Uma das principais melhorias introduzidas pelo Ext2 foi a possibilidade de configurar a densidade de *i-nodes* durante a formatação do sistema de arquivos. Diferentemente do Ext original, em que a quantidade de *i-nodes* era definida e não podia ser ajustada, o Ext2 permite ao administrador determinar o número máximo de *i-nodes* com base no perfil de uso da partição. Essa configuração permanece fixa após a criação do sistema de arquivos, mas oferece flexibilidade para adequar a estrutura tanto a ambientes com grande quantidade de arquivos pequenos quanto a volumes destinados a arquivos extensos. A opção de definir o tamanho de bloco entre 1.024 e 4.096 bytes complementa esse ajuste, permitindo equilibrar desempenho e eficiência de armazenamento ao reduzir a fragmentação interna e otimizar o custo das operações de leitura e escrita.

Cada *i-node* contém um conjunto de ponteiros diretos que referenciam blocos de dados de forma imediata, adequado para arquivos de pequena extensão. À medida que o arquivo cresce, o sistema recorre a níveis adicionais de indireção: o ponteiro indireto simples referencia um bloco que contém apenas endereços de blocos de dados; o indireto duplo aponta para um bloco cujas entradas, por sua vez, apontam para blocos de indireção simples; e o indireto triplo encadeia mais um nível nessa hierarquia. Esse esquema permite que arquivos potencialmente grandes sejam representados sem aumentar o tamanho fixo do *i-node*, mas introduz um custo adicional de acesso, especialmente quando múltiplos níveis precisam ser percorridos para localizar os dados.

O Ext2 organiza os dados em grupos de blocos, cada um contendo seu próprio superbloco, *bitmaps* e tabela de *i-nodes*. Essa segmentação melhora a localidade espacial dos blocos no disco e reduz o tempo de busca, uma vez que os metadados e os blocos de dados de um mesmo arquivo tendem a residir próximos no disco. Esse modelo de agrupamento foi projetado especificamente para discos magnéticos, nos quais a movimentação da cabeça de leitura representava uma parcela significativa do tempo de acesso. Em unidades de estado sólido, esse benefício é menos relevante,

mas a organização modular continua favorecendo a escalabilidade e a integridade estrutural do sistema.

O sistema também adota a pré-alocação de blocos de dados como forma de reduzir a fragmentação externa e melhorar o desempenho em acessos sequenciais. Essa técnica reserva blocos contíguos no momento da criação de arquivos, antecipando futuras expansões e garantindo que os dados sejam gravados em regiões fisicamente próximas. Essa abordagem é vantajosa em aplicações que manipulam fluxos contínuos de dados, como bancos de dados ou arquivos multimídia.

Outro conceito fundamental para o entendimento do Ext2 é o de *links*, mecanismos que permitem associar múltiplos nomes a um mesmo arquivo. O sistema distingue entre dois tipos: *hard links* e *symbolic links*. Os *hard links* criam referências diretas ao mesmo *i-node*, de modo que diferentes nomes de arquivo apontam para o mesmo conteúdo armazenado em disco. Já os *links* simbólicos armazenam apenas o caminho de destino, funcionando como ponteiros que o sistema resolve em tempo de acesso.

A resolução de *links* simbólicos foi otimizada para reduzir o número de acessos a disco. Em outros sistemas de arquivos, o caminho de destino de um *link* simbólico é armazenado em um bloco de dados comum, exigindo uma leitura adicional sempre que o sistema precisa interpretar o *link*. O Ext2 evita essa operação ao registrar o caminho diretamente no *i-node*, desde que o conteúdo possua até 60 *bytes*. Essa estratégia elimina a necessidade de buscar o bloco de dados associado ao *link*, reduzindo o número de operações de entrada e saída (*I/O)* e otimizando o processo de resolução de caminhos.

O sistema mantém a consistência estrutural dos arquivos por meio de um contador de referências associado a cada *i-node*. Esse contador registra o número de *hard links* existentes, ou seja, de nomes distintos que apontam para o mesmo arquivo. Quando um novo *hard link* é criado, o contador é incrementado; quando um nome é removido, ele é decrementado. O conteúdo de um arquivo permanece acessível enquanto o contador for maior que zero, garantindo que os blocos de dados associados não sejam liberados. Essa abordagem assegura a integridade das referências sem exigir estruturas auxiliares, preservando a coerência entre diretórios e *i-nodes* mesmo em operações de atualização simultânea.

3.2.1 Grupo de Blocos

A Figura 3.1 apresenta a organização de uma partição formatada com Ext2 em um dispositivo. Após o bloco de inicialização, a partição é dividida em múltiplos grupos de blocos, dispostos de forma sequencial e de tamanho uniforme. Essa segmentação permite ao sistema localizar diretamente o início de qualquer grupo por meio de cálculos baseados em seu índice.

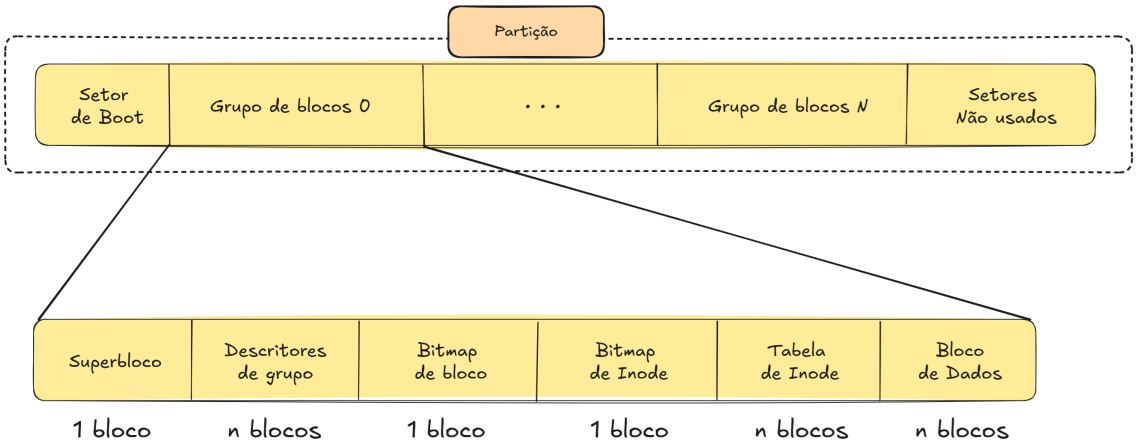


Figura 3.1: Distribuição contígua de blocos com 1 arquivo excluído.

Cada grupo de blocos reúne as estruturas necessárias ao gerenciamento local do sistema de arquivos. O superbloco e os descritores de grupo possuem cópias distribuídas entre os grupos, contendo informações globais sobre o sistema de arquivos e referências às estruturas internas de cada grupo. Em seguida localizam-se os *bitmaps* de blocos e de *i-nodes*, responsáveis por indicar quais unidades estão livres ou alocadas.

A tabela de *i-nodes* do grupo contém as estruturas de metadados dos arquivos, incluindo permissões, tamanho, marcações de tempo e ponteiros para os blocos de dados. O espaço restante do grupo é destinado aos blocos de dados propriamente ditos, cujo conteúdo é acessado a partir dos ponteiros armazenados nos *i-nodes*. A organização em grupos busca manter arquivos relacionados próximos entre si, reduzindo movimentações de leitura em dispositivos mecânicos e favorecendo padrões de acesso sequencial.

Como mostra a Figura 3.1, as cópias de superbloco e dos descritores de grupo são mantidas em todos os grupos, mas apenas as localizadas no grupo 0 são utilizadas rotineiramente pelo *kernel*. Durante verificações de integridade, o utilitário `e2fsck` utiliza esses metadados do primeiro grupo de blocos e pode reconstruir as cópias dos demais grupos conforme necessário [3].

O número de grupos de blocos que compõem um dispositivo com Ext2 depende de seu tamanho total e do tamanho de bloco adotado. Essa divisão busca equilibrar a quantidade de *i-nodes*, o tamanho dos *bitmaps* e a proximidade física entre metadados e dados, contribuindo para acessos mais previsíveis e para a distribuição uniforme da carga de alocação.

3.2.2 Exclusão de Arquivos

A remoção de arquivos no Ext2 não apaga imediatamente os dados armazenados em disco. O procedimento consiste, inicialmente, na remoção da entrada correspondente

no diretório e na redução do contador de *hard links* presente no *i-node*. Somente quando esse contador atinge zero o *i-node* é considerado não referenciado, permitindo que o sistema marque seus blocos de dados como livres no *bitmap*. Dessa forma, embora o conteúdo permaneça fisicamente no disco até ser sobrescrito por novas alocações, ele deixa de ser acessível por meio da estrutura de diretórios.

A ordem em que essas atualizações são realizadas é relevante para preservar a consistência do sistema de arquivos. Caso a remoção do *i-node* ocorresse antes da atualização do diretório, uma falha durante a operação poderia resultar em uma entrada de diretório apontando para um *i-node* inválido. Se esse *i-node* viesse a ser posteriormente reutilizado para outro arquivo, a entrada residual no diretório passaria a referenciar metadados incorretos, potencialmente levando à sobrescrita acidental de dados ou à leitura de informações incorretas. A abordagem implementada no Ext2 reduz o risco de inconsistências estruturais ao garantir que o diretório seja atualizado somente após o estado do *i-node* ter sido registrado de forma coerente.

3.3 Ext3

O *Third Extended File System* (Ext3) foi desenvolvido com o objetivo de resolver uma limitação do Ext2: a ausência de um mecanismo de registro de operações capaz de garantir a consistência do sistema após falhas inesperadas como, por exemplo, desligamentos inesperados ou falhas no sistema operacional. A fim de tornar o sistema mais robusto foi implementada a estratégia de *journaling*. Além disso, sua arquitetura manteve compatibilidade total com o Ext2, permitindo a migração de sistemas legados sem necessidade de reformatação.

3.3.1 Mecanismo de *Journaling*

Em sistemas que não possuem *journaling*, uma interrupção durante operações de escrita, como queda de energia ou travamento do sistema operacional, pode deixar o sistema em estado inconsistente, exigindo a execução de uma verificação completa do disco no próximo *boot*. Esse procedimento é custoso e pode levar minutos ou horas, dependendo do tamanho da partição ou do disco.

O Ext3 introduziu o *journaling* de metadados para mitigar esse problema. Antes de gravar alterações na estrutura do sistema de arquivos, as operações são registradas em uma área de *log* denominada *journal*. Em caso de falha, o sistema não precisa verificar toda a partição: basta identificar transações incompletas no *journal* e reaplicá-las para restaurar o sistema a um estado consistente [18].

Essa abordagem reduz o tempo de recuperação e aumenta a robustez quando falhas inesperadas ocorrem. Em cenários específicos, o mecanismo também pode melhorar o desempenho, eliminando a necessidade de protocolos complexos de consistência e aproveitando técnicas como somas de verificação para confirmar a inte-

gridade das transações.

3.3.2 Indexação de Diretórios

Outra limitação herdada pelo Ext3 estava relacionada ao método de organização das entradas de diretório. Em sua versão anterior, os diretórios eram representados como listas lineares, de modo em que operações como busca, inserção ou remoção exigiam a leitura sequencial de cada entrada. O custo dessa operação crescia proporcionalmente ao número de arquivos presentes no diretório e tornava-se ineficiente em cenários com centenas ou milhares de entradas, nos quais o tempo de resposta aumentava de forma significativa [6].

Para mitigar esse problema, o Ext3 implementou a estrutura denominada *HTree* (*Hashed Tree*), um mecanismo de indexação baseado em princípios de árvores B, porém adaptado ao contexto de diretórios. O HTree utiliza um valor de dispersão (*hash*) calculado a partir do nome de cada arquivo como chave para indexação, permitindo organizar as entradas em blocos internos responsáveis pelo direcionamento das buscas e em blocos que armazenam as entradas reais. Essa segmentação permite que o diretório seja percorrido de maneira hierárquica, reduzindo a necessidade de varreduras lineares.

A profundidade fixa adotada pela estrutura, limitada a dois níveis, garante que as operações de busca, inserção e exclusão apresentem complexidade próxima de $O(log\ n)$, independentemente do número de arquivos no diretório [7]. Como resultado, o Ext3 supera o custo linear associado ao modelo de lista sequencial e oferece um mecanismo de acesso escalável, adequado para diretórios com grande quantidade de arquivos e cargas de trabalho intensivas.

3.4 Ext4

O Ext4 foi projetado como sucessor do Ext3, com o objetivo de superar limitações relacionadas à escalabilidade, fragmentação e desempenho dos sistemas anteriores. Embora mantenha compatibilidade estrutural com Ext2/Ext3, o Ext4 implementa alterações em seu modelo de endereçamento e no gerenciamento de metadados.

3.4.1 Endereçamento por *Extents*

Uma das modificações implementadas pelo Ext4 é a adoção de *Extents* em substituição ao esquema de blocos indiretos utilizado pelo Ext2/Ext3. No novo modelo, um *Extent* representa um intervalo contíguo de blocos físicos, descrito por meio de um único registro contendo o bloco lógico inicial, o bloco físico inicial e o comprimento do intervalo. Essa substituição reduz o volume de metadados necessários para representar arquivos e elimina a sobrecarga imposta pelos múltiplos níveis de indireção do esquema anterior.

A estrutura de *Extents* é organizada como uma Árvore B+, denominada *Extent Tree*, cuja raiz é armazenada diretamente no *i-node* de cada arquivo. Os nós internos da árvore contêm registros do tipo *Extent-Index*, responsáveis por apontar para níveis inferiores, enquanto os nós folha armazenam os *Extents* propriamente ditos, que mapeiam os blocos físicos correspondentes. A navegação nessa árvore é realizada por meio de busca binária em cada nó, garantindo um caminho de acesso eficiente e assintoticamente estável entre o bloco lógico solicitado e seu respectivo bloco físico.

A Figura 3.2 mostra a organização hierárquica da *Extent Tree*, mostrando a relação entre o *i-node*, os nós internos e os nós folha que apontam diretamente para os blocos de dados.

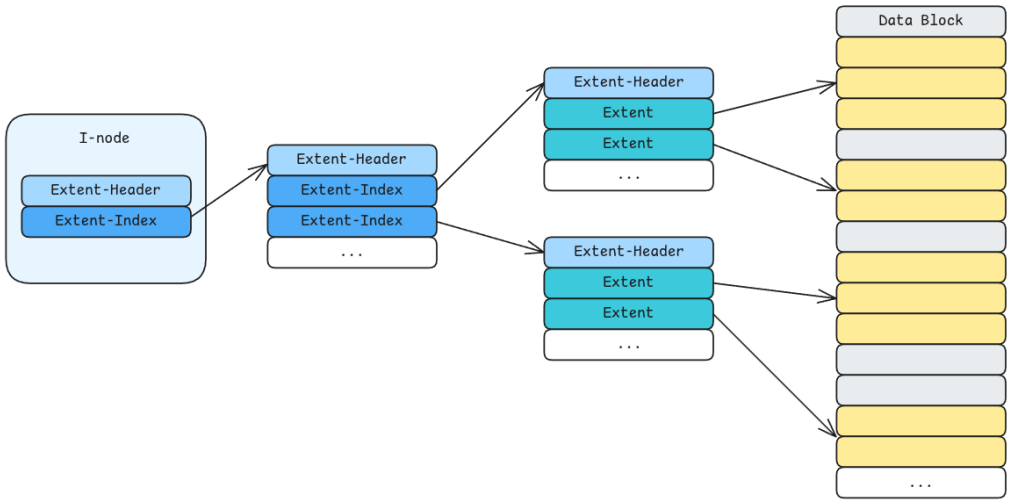


Figura 3.2: Estrutura hierárquica da *Extent Tree* no Ext4 [16].

3.4.2 Melhorias de Desempenho

Com o objetivo de mitigar problemas de fragmentação e otimizar o desempenho de escrita, o Ext4 implementa a técnica de alocação atrasada, que posterga a alocação física dos blocos até o instante efetivo da escrita no dispositivo de armazenamento. Ao adiar essa decisão, o alocador passa a dispor de informações mais completas sobre o padrão de escrita, permitindo a seleção de áreas contíguas e favorecendo a criação de *Extents* mais longos. Esse mecanismo reduz a probabilidade de fragmentação interna e externa, melhora o *throughput* e diminui a sobrecarga de operações de entrada e saída (I/O).

Como complemento, a alocação múltipla de blocos permite a reserva simultânea de conjuntos de blocos contíguos, otimizando a construção de arquivos sequenciais e reduzindo o custo computacional associado à alocação bloco a bloco.

Outras melhorias incluem o aumento do tamanho padrão do *i-node* para 256 *bytes*, permitindo armazenar metadados adicionais, como *timestamps* com resolução

de nanossegundos, além de mecanismos de *checksumming* aplicados ao *journaling*, com o objetivo de ampliar a robustez contra corrupção de metadados. Por fim, o recurso de grupos não-inicializados possibilita ao utilitário de verificação ignorar grupos de *i-nodes* ainda não inicializados, reduzindo o tempo necessário para a verificação de consistência do sistema de arquivos.

3.5 *Copy-on-Write*

O *Copy-on-Write* (COW) é um princípio arquitetural empregado por sistemas de arquivos como o ZFS e Btrfs, e consiste na atualização de dados e metadados sem sobrescrever diretamente seus conteúdos já persistidos. Em uma operação de escrita, os blocos modificados são registrados em novas regiões do dispositivo de armazenamento. Somente após a conclusão dessa escrita é que as estruturas de metadados são atualizadas para referenciar os novos blocos.

Como consequência direta desse mecanismo, qualquer modificação em uma folha da árvore de metadados, normalmente organizada como variações de uma Árvore B, requer a atualização dos nós internos que apontam para ela. Esses nós também são gravados em novos locais, pois seus ponteiros passam a referenciar blocos distintos. Esse processo se propaga até a raiz, produzindo uma nova cadeia de metadados que representa a versão atual do sistema, enquanto a versão anterior permanece intacta até que a atualização da raiz seja concluída de forma atômica.

Embora garanta alta consistência, o mecanismo de *copy-on-write* causa um custo adicional ao desempenho, pois cada modificação exige a criação de um novo bloco e a atualização dos ponteiros ao longo da árvore. Em operações de escrita intensivas, esse processo de copiar antes de escrever aumenta o tempo de de entrada e saída em comparação com atualizações realizadas diretamente no bloco original.

Essa abordagem reduz o risco de inconsistência estrutural. Caso ocorra uma falha durante a operação, o sistema continua a referenciar a versão anterior dos dados e metadados, que permanece válida e completa [23]. A estratégia de COW cumpre o papel que, em sistemas não o utilizam, é desempenhado pelo *journaling*: preservar a consistência dos metadados após falhas abruptas.

O mesmo princípio possibilita a criação eficiente de *snapshots*. Como os dados não são sobrescritos, o sistema mantém múltiplas raízes que referenciam diferentes versões de uma mesma árvore de metadados. Essas versões compartilham todos os blocos que não foram alterados, de modo que a criação de um *snapshot* consiste apenas no registro de uma nova raiz, operação que não exige cópia imediata de dados [15]. Quando uma das versões é modificada, apenas os blocos afetados e seus metadados ancestrais são alocados novamente, formando uma nova cadeia de atualizações. Essa técnica também permite a criação de clones mutáveis, que evoluem independentemente do volume de origem.

Embora o COW proporcione propriedades relevantes de integridade e versiona-

mento, também introduz um efeito conhecido como amplificação de escrita. Como alterações localizadas podem exigir a regravação de múltiplos níveis da árvore de metadados, a quantidade total de blocos modificados por operação pode ser maior do que em sistemas que empregam sobrescrita direta [15].

3.6 ZFS

O ZFS (*Zettabyte File System*), desenvolvido originalmente pela Sun Microsystems, introduz uma arquitetura integrada que combina o sistema de arquivos e o gerenciamento de volumes em uma única camada. Essa integração contrasta com modelos tradicionais, como os utilizados pelo Ext4, nos quais a organização lógica do volume e o sistema de arquivos operam como componentes separados. No ZFS, essa distinção é eliminada, permitindo um controle global sobre dispositivos físicos, alocação de espaço, redundância e integridade dos dados [23].

O sistema implementa um modelo de escrita baseado em *Copy-on-Write* (COW), no qual dados e metadados modificados são gravados em novos blocos, preservando-se intacta a versão anterior até a conclusão da operação. Esse mecanismo elimina a necessidade de estruturas tradicionais de *journaling* para consistência dos metadados e reduz o risco de corrupção em cenários de falhas abruptas [23].

Além do COW, o ZFS adota verificação de integridade de ponta a ponta por meio de *checksums* aplicados tanto a dados quanto a metadados. Cada bloco armazenado possui uma soma de verificação registrada em seus metadados superiores. Durante leituras, o ZFS recalcula o *checksum* do bloco e o compara ao valor registrado, permitindo detectar e, quando possível, corrigir corrupção silenciosa ao utilizar cópias redundantes disponíveis [10]. Essa propriedade confere ao sistema um nível de confiabilidade superior ao de sistemas que verificam apenas metadados ou utilizam técnicas de *journaling*.

Embora utilize o conceito de intervalos contíguos de blocos (*extents*) para representar dados, o ZFS não utiliza Árvore B+. Em vez disso, suas estruturas internas são organizadas pela *Data Management Unit* (DMU), que mantém árvores baseadas em COW onde cada nó contém ponteiros para blocos de dados ou metadados, acompanhados de identificadores de transação. Essa organização permite que múltiplas versões de uma mesma estrutura coexistam, viabilizando operações como *snapshots* e clonagem sem duplicação imediata de dados.

A combinação do modelo COW com a organização interna das árvores da DMU permite ao ZFS criar *snapshots* de modo eficiente. Um *snapshot*, no ZFS, consiste em um ponto estável da árvore de metadados, formado pelo registro de uma nova raiz. Como os blocos não modificados são compartilhados entre as versões, a criação de um *snapshot* não acarreta cópia de dados. Além disso, o ZFS disponibiliza mecanismos de replicação incremental, nos quais apenas as diferenças entre dois *snapshots* são transmitidas, otimizando processos de recuperação e sincronização entre sistemas.

3.6.1 Gerenciamento Unificado de Armazenamento

O gerenciamento de armazenamento no ZFS é estruturado por meio de *storage pools* (ou *zpools*). Um *pool* representa um agrupamento lógico de dispositivos de armazenamento, organizados em estruturas chamadas *vdevs* (*virtual device* — dispositivo virtual). Cada *vdev* pode ser configurado com diferentes níveis de redundância, como espelhamento ou RAID-Z, e o conjunto desses dispositivos é tratado como uma única fonte de blocos para o sistema de arquivos.

Essa abordagem substitui a necessidade de camadas externas de gerenciamento de volumes, como LVM ou `mdadm`[1], e permite ao ZFS efetuar a alocação de espaço, o balanceamento interno e a distribuição de dados entre dispositivos de forma autônoma. A consistência global do *pool* é assegurada por um conjunto de *uberblocks*, estruturas que funcionam como pontos de entrada para o estado atual do sistema. Cada *uberblock* contém informações de transação, é protegido por *checksums* e é atualizado de maneira atômica, garantindo que sempre exista uma versão consistente do *pool* acessível após falhas.

3.7 Btrfs

O Btrfs (*B-tree file system*) é um sistema de arquivos para Linux cujo projeto combina o uso de árvores B balanceadas e a política de escrita *Copy-on-Write* (COW). Em contraste com sistemas como Ext4, que dependem de tabelas de *i-nodes* pré-alocadas, o Btrfs organiza seus metadados em árvores especializadas atualizadas dinamicamente conforme o volume cresce, se fragmenta ou cria novas versões.

No Btrfs, todo o estado persistente é representado por árvores B. O sistema mantém múltiplas árvores, cada uma responsável por uma categoria distinta de metadados: a árvore de arquivos (`FS tree`) armazena *i-nodes* e diretórios; a árvore de *extents* descreve o mapeamento entre endereços lógicos e blocos físicos; a árvore de *chunks* registra a disposição física do volume; e a árvore de *checksums* contém os valores de verificação associados aos blocos. Todas seguem o mesmo formato geral, utilizando chaves compostas e operações de busca, inserção e remoção com custo computacional de $O(log\ n)$ [4].

Diferentemente de sistemas baseados em tabelas estáticas, como Ext4, essa modelagem dispensa a pré-alocação de regiões destinadas a metadados. Dessa forma, novos *i-nodes*, diretórios e *extents* podem ser criados conforme necessário, o que reduz a dependência entre tamanho do volume e espaço reservado para metadados e contribui para a escalabilidade do sistema.

O Btrfs utiliza *extents* para representar sequências contíguas de blocos associados a arquivos ou metadados. Ao modificar um bloco pertencente a um *extent*, o mecanismo COW grava o conteúdo atualizado em um novo local e ajusta as árvores

[1] O LVM (Logical Volume Manager) e o `mdadm` são ferramentas do Linux para gerenciamento lógico de volumes e configuração de arranjos RAID em camadas separadas do sistema de arquivos.

correspondentes, gerando novos nós desde a folha até a raiz. Esse procedimento preserva as versões anteriores até a finalização da operação e viabiliza a criação eficiente de *snapshots* e clones [15]. A ausência de sobrescrita no Btrfs facilita o versionamento, mas pode aumentar a quantidade total de metadados modificados em situações que exigem sincronização frequente.

O Btrfs organiza seu espaço lógico por meio de *subvolumes*, unidades independentes que possuem sua própria árvore de arquivos. Um *subvolume* pode ser representado por meio de um *snapshot*, que consiste na criação de uma nova raiz para a mesma estrutura de metadados existente. Como blocos não modificados são compartilhados, a criação de *snapshots* é imediata e não demanda cópia de dados [15].

A integridade é assegurada por meio de *checksums* armazenados na árvore dedicada a esse propósito. Em cada leitura, o sistema recalcula o valor de verificação do bloco e compara-o ao valor registrado. Havendo redundância disponível, o Btrfs é capaz de recuperar blocos corrompidos, reforçando sua resiliência estrutural.

A combinação de árvores B dinâmicas, *extents* e escrita baseada em COW confere ao Btrfs flexibilidade para operações de clonagem e criação de *snapshots*. Embora esse modelo introduza custos adicionais em cargas que exigem sincronização frequente, oferece mecanismos nativos de versionamento e verificação de integridade integrados ao próprio sistema de arquivos.

3.8 Comparação entre Sistemas de Arquivos

A partir da revisão apresentada neste capítulo, é possível sintetizar algumas características estruturais dos sistemas de arquivos analisados, com foco em quatro aspectos: forma de indexação de metadados, mecanismo de endereçamento de dados, estratégias de integridade em caso de falhas e presença ou ausência de recursos nativos de recuperação de arquivos excluídos. A Tabela 3.1 resume essas características para os sistemas Ext2, Ext3, Ext4, ZFS e Btrfs, bem como para o *Basic Solution File System* (BSFS), cuja arquitetura será detalhada no Capítulo 4.

Tabela 3.1: Comparação entre características de sistemas de arquivos e o BSFS.

Sistema	**Indexação**	**Endereçamento**	**Integridade**	**Recup. Nativa**
Ext2	Tabela de *i-nodes*	Indireção múltipla	*fsck*	Não
Ext3	Tabela de *i-nodes* e *HTree*	Indireção múltipla	*Journaling*	Não
Ext4	Tabela de *i-nodes* e *HTree*	*Extents*	*Journaling*	Não
ZFS	Árvores B+	*Extents*	COW e *checksums*	Não
Btrfs	Árvores B+	*Extents*	COW e *checksums*	Não
BSFS	Árvores B	*Bspan*	Não apresenta	Sim

3.9 Sumário

Este capítulo apresentou a evolução dos sistemas de arquivos utilizados em ambientes Unix-*like*, destacando como decisões de projeto distintas buscaram equilibrar desempenho, escalabilidade, integridade e simplicidade de administração. A discussão iniciou-se com os sistemas de arquivos clássicos do Unix e com o Berkeley Fast File System (FFS), que consolidaram conceitos como *i-nodes*, diretórios hierárquicos, grupos de blocos e estratégias de gerenciamento de espaço livre, servindo de base para os sistemas posteriores.

Em seguida, foram analisados os sistemas da família Ext. O Ext2 foi descrito como a primeira solução amplamente adotada no Linux, incorporando grupos de blocos, configuração de densidade de *i-nodes*, escolha do tamanho de bloco e mecanismos de pré-alocação, além do tratamento de *links* e da ordem das operações de exclusão. O Ext3 foi apresentado como uma evolução compatível, que introduziu *journaling* de metadados para acelerar a recuperação após falhas e passou a empregar a indexação de diretórios por *HTree*. O Ext4, por sua vez, substituiu o esquema de blocos indiretos por *extents* organizados em uma *Extent Tree*, ampliou o espaço de endereçamento e incorporou técnicas de alocação atrasada e multibloco, além de melhorias em metadados e mecanismos de verificação, aproximando-se das demandas de discos de maior capacidade.

Além disso, o capítulo apresentou o modelo de *Copy-on-Write* (COW), utilizado pelos sistemas de arquivos ZFS e Btrfs. Foram descritos o modo de atualização sem sobrescrita, a propagação de modificações nas árvores de metadados, as garantias de consistência estrutural e o papel desse modelo na viabilização de *snapshots* e clones. Também foram discutidos os efeitos de amplificação de escrita associados a essa técnica.

Na sequência, foram examinados dois sistemas contemporâneos que adotam COW: ZFS e Btrfs. O primeiro foi descrito como uma arquitetura que integra o gerenciamento de volumes e o sistema de arquivos em *storage pools* organizados em *vdevs*, empregando escrita baseada em COW, verificação de integridade de ponta a ponta por meio de *checksums*, estruturas mantidas pela *Data Management Unit* (DMU) e mecanismos de *snapshots* e replicação incremental. O segundo foi analisado como uma solução moderna para Linux que representa praticamente todo o estado persistente em múltiplas B-trees dinâmicas, incluindo as árvores de arquivos, de *extents*, de *chunks* e de *checksums*, combinando *extents* com COW e oferecendo recursos como subvolumes, *snapshots* e verificação de integridade integrada.

Em conjunto, esses sistemas evidenciam três tendências principais: a adoção de índices hierárquicos em lugar de estruturas lineares, o uso de *extents* para substituir múltiplos níveis de indireção e a inclusão de mecanismos de integridade, como *journaling* ou COW com *checksums*. Essas tendências também orientam o projeto do BSFS, que utiliza intervalos contíguos de blocos e estruturas indexadas para organizar seus metadados, conforme detalhado no Capítulo 4.

Capítulo 4

Basic Solution File System

> *Code is not like other how-computers-work books. It doesn't have big color illustrations of disk drives with arrows showing how the data sweeps into the computer. Code has no drawings of trains carrying a cargo of zeros and ones. Metaphors and similes are wonderful literary devices but they do nothing but obscure the beauty of technology.*
>
> – Charles Petzold

Neste capítulo, serão abordadas as características de implementação do *Basic Solution File System* (BSFS), um sistema de arquivos desenvolvido em linguagem C, executado em espaço de usuário no ambiente Linux, cuja principal funcionalidade é o mecanismo de recuperação de arquivos previamente excluídos. Inicia-se tratando da estrutura que contém as principais informações sobre uma partição formatada com o sistema de arquivos, o superbloco. Em seguida, aborda a decisão de projeto de escolha do tamanho do bloco e o mapeamento do espaço livre.

Para facilitar o entendimento das estruturas de dados utilizadas no projeto, é apresentada a conceitualização da árvore B, estrutura fundamental para o gerenciamento de *i-nodes*, o endereçamento de blocos, o funcionamento dos diretórios e o processo de recuperação de arquivos.

Após a definição das estruturas de dados e operações fundamentais, o capítulo apresenta a implementação da interface de interação com o usuário, responsável por expor as funcionalidades do BSFS por meio de comandos e utilitários específicos. Em seguida, são discutidos os procedimentos de teste adotados e os resultados obtidos em operações de gravação de arquivos, permitindo comparar o comportamento do BSFS com sistemas de arquivos consolidados em ambientes Linux.

4.1 Superbloco

O superbloco do BSFS segue os princípios descritos na Seção 2.1, atuando como a principal estrutura de controle e identificação. É armazenado em uma posição fixa no primeiro bloco da partição e carregado em memória sempre que o BSFS é utilizado. A estrutura *superblock_t* define esse componente, reunindo informações sobre como o sistema está configurado e onde estão as outras estruturas. O campo `magic_number` armazena o valor constante para identificar o sistema de arquivos, e os campos `block_size` e `fs_size` definem, respectivamente, o tamanho de cada bloco físico em *bytes* e o tamanho total da partição formatada. Já o campo `total_blocks` indica o número total de blocos disponíveis na partição, servindo como base para o cálculo do espaço livre e para a inicialização das rotinas de alocação.

Em seguida, o superbloco registra os campos que descrevem a organização lógica do sistema de arquivos. Os campos `block_bitmap_start` e `block_bitmap_total` indicam, respectivamente, o bloco inicial e a quantidade de blocos utilizados pelo mapa de *bits* responsável pelo controle de espaço livre. Os campos `inode_root` e `recovery_root` armazenam os blocos que contêm as raízes das duas árvores B do sistema: a primeira dedicada à indexação dos *i-nodes* e a segunda ao gerenciamento das entradas de recuperação. Por fim, `data_block_start` define o início da área de dados dos arquivos, enquanto `root_inode` registra o *i-node* correspondente ao diretório raiz.

Essa organização foi projetada para permitir o acesso direto às principais estruturas do BSFS com o mínimo de operações de leitura em disco. Como o sistema de arquivos é executado em espaço de usuário, o superbloco é lido e interpretado diretamente pela aplicação abordada na Seção 4.8, sem intermediação do *kernel*. A definição explícita dos endereços e tamanhos de cada região garante acesso às estruturas do BSFS a partir de uma consulta única ao superbloco durante as operações. Dessa forma, o superbloco atua como um mapa fixo do sistema, centralizando todas as informações necessárias para localizar e manipular as demais estruturas em disco.

4.2 Gerenciamento do Espaço Livre

O BSFS adota blocos de 4 KiB como unidade de alocação, valor definido com base nos critérios discutidos na Seção 2.2.3. A escolha oferece um equilíbrio entre desempenho e eficiência de espaço, além de alinhar-se ao tamanho de página comumente utilizado pelos sistemas operacionais, simplificando o gerenciamento de *buffers* e as operações de entrada e saída em espaço de usuário. O uso de blocos de 4 KiB reduz a fragmentação interna observada em blocos maiores e, ao mesmo tempo, evita a sobrecarga de leitura e escrita que ocorreria com blocos menores.

Para rastrear blocos livres e ocupados, o BSFS implementa um *bitmap* armazenado em disco, segmentado em blocos de *bitmap* e organizado como uma lista encadeada, conforme mostrado na Figura 4.1. Cada *bit* representa um bloco de

dados: 0 (*livre*) e 1 (*ocupado*), convenção adotada pelo BSFS, e cada nó da lista corresponde a uma estrutura lógica *bitmap_block_t*, composta por um vetor de *bits* e um ponteiro `next_block` para o próximo bloco do *bitmap*.

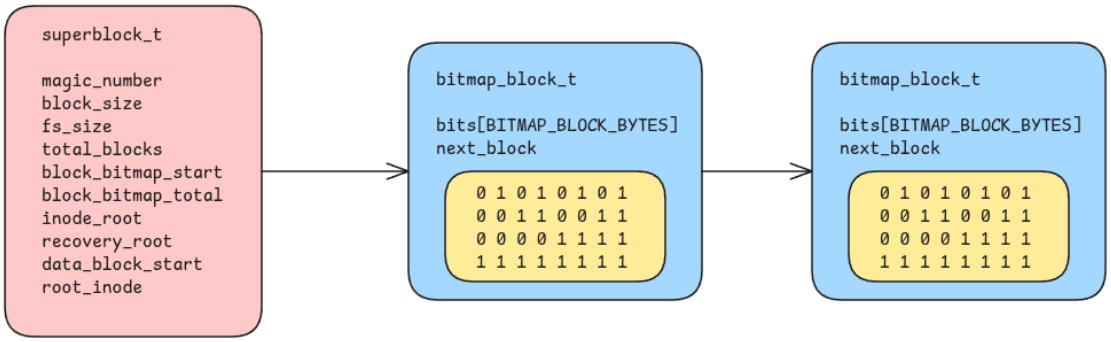


Figura 4.1: Lista ligada do *bitmap* implementado no BSFS.

Os campos `block_bitmap_start` e `block_bitmap_total` do superbloco delimitam a região inicial e a capacidade alocada para o conjunto de blocos de *bitmap*, enquanto `data_block_start` marca o início da área de dados. Dessa forma, o *bitmap* pode crescer de forma incremental, adicionando novos nós à lista sem mover estruturas existentes.

Durante a escrita de arquivos, a alocação de novos blocos segue a estratégia *first-fit*, na qual o sistema percorre sequencialmente os blocos do *bitmap* até encontrar o primeiro intervalo de *bits* livres capaz de atender à quantidade de blocos solicitados. Assim que um espaço adequado é identificado, os *bits* correspondentes são marcados como ocupados no mapa, e seus números de bloco são retornados para gravação. Essa abordagem foi escolhida pela simplicidade de implementação em relação a outros métodos mais sofisticados, como *best-fit* e *quick-fit*.

4.3 Árvore B

As árvores B são estruturas de dados de busca balanceada projetadas para operar de forma eficiente em dispositivos de armazenamento secundário, como discos rígidos e unidades de estado sólido. Diferentemente das árvores binárias tradicionais, cada nó pode conter múltiplas chaves e vários filhos, reduzindo a altura da árvore e, consequentemente, o número de acessos ao disco necessários em operações de busca e inserção. Essa característica torna esse tipo de estrutura de dados ideal para sistemas de arquivos, em que o custo de leitura e escrita em disco é maior do que operações realizadas na memória principal [5].

Os sistemas de arquivos abordados no Capítulo 3 utilizam variações de árvores B (Ext4, ZFS e Btrfs), nas quais apenas as folhas armazenam os metadados dos arquivos, enquanto os nós internos contêm chaves e endereços de blocos. No BSFS,

optou-se pela implementação de uma árvore B tradicional, em que os metadados são armazenados diretamente nos nós internos. Essa estrutura de dados foi implementada integralmente para o projeto, sem o uso de bibliotecas externas, a fim de permitir controle total sobre o formato dos nós, a política de persistência e o modelo de interação com o armazenamento secundário. A decisão tem como objetivo reduzir o número total de blocos ocupados pela estrutura, ainda que as operações de busca, inserção e remoção apresentem um custo computacional maior. O equilíbrio entre simplicidade estrutural e economia de espaço motivou essa escolha no contexto do projeto.

No BSFS, a árvore B foi implementada para operar sobre o armazenamento secundário, seguindo o modelo de operações `DISK-READ` e `DISK-WRITE` [5], que abstraem o processo de leitura e gravação de páginas de dados entre a memória principal e o disco. Cada nó da árvore ocupa um bloco inteiro da partição, contendo metadados e ponteiros para blocos-filhos. A estrutura foi projetada para uso de quatro tipos diferentes de dados como chave; as Seções 4.4 à 4.7 detalham as estruturas utilizadas na implementação.

A implementação da árvore B no BSFS contempla as principais operações descritas na literatura: criação, busca, inserção, remoção e atualização de chaves. O objetivo central dessas rotinas é permitir a manipulação de conjuntos de metadados, garantindo a integridade e a consistência da árvore após cada modificação.

A operação de busca segue o princípio da busca em árvores de pesquisa binária, porém com múltiplos caminhos possíveis a partir de cada nó. Ao receber uma chave, o algoritmo percorre sequencialmente as chaves armazenadas no nó atual até determinar o intervalo correspondente e, então, acessa o nó-filho adequado. No BSFS, cada acesso a um nó envolve uma leitura de bloco em disco, e o processo continua até que a chave seja localizada ou que uma folha seja alcançada. Assim como no modelo teórico, a complexidade da busca é proporcional à altura da árvore, mantendo-se em ordem logarítmica mesmo para grandes quantidades de registros.

A inserção de novos elementos é executada de forma descendente, garantindo que nenhum nó exceda o número máximo de chaves permitido. Quando um nó fica completo, ocorre sua divisão, na qual a chave intermediária sobe para o nó pai e as chaves remanescentes são distribuídas entre dois novos blocos. Esse procedimento mantém o balanceamento da árvore e preserva suas propriedades estruturais. No BSFS, a divisão de nós envolve tanto a alocação de novos blocos quanto a atualização de ponteiros armazenados, o que torna o processo mais custoso que em implementações puramente em memória, porém essencial para garantir persistência e integridade dos metadados.

A remoção de registros segue um processo similar à inserção, com o objetivo de evitar nós com menos chaves do que o grau mínimo permitido. Quando um nó fica abaixo do limite, ele é combinado com um nó irmão ou recebe uma chave emprestada de um nó vizinho, de modo a restabelecer as propriedades da árvore.

A operação de atualização é tratada como uma substituição da chave existente, sem necessidade de reestruturação, salvo em casos de realocação de blocos. Todas as operações são gravadas em armazenamento secundário por meio de chamadas às funções de leitura e escrita de blocos, assegurando a persistência das alterações.

4.4 Implementação de *I-nodes*

A estrutura de *i-node* utilizada no BSFS segue o princípio funcional descrito na Seção 2.2.1, atuando como o principal repositório de metadados de cada arquivo e diretório. O *i-node* armazena identificadores, permissões, tamanho do arquivo, marcações de tempo de criação e modificação, e referências para os blocos de dados. A estrutura constitui um dos tipos genéricos aceitos pela árvore B, como mostrado na Figura 4.2.

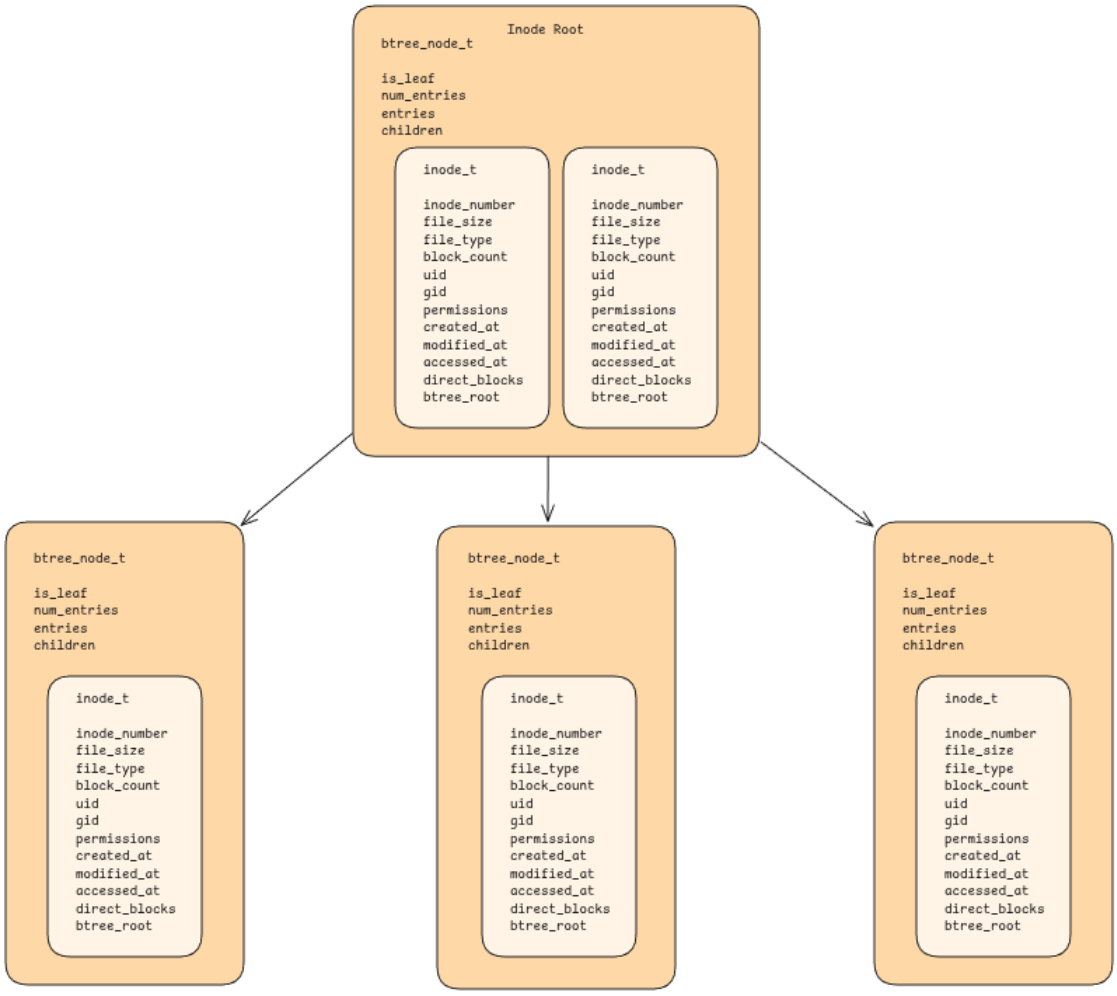


Figura 4.2: Exemplo de árvore B de *i-nodes* implementada no BSFS.

Cada *i-node* é identificado por um número único (`inode_number`), que serve

como chave na árvore B responsável pela indexação dos metadados. A árvore é armazenada em disco e possui como raiz o bloco referenciado pelo campo `inode_root` do superbloco. As operações de criação, leitura, atualização e remoção são implementadas como modificações diretas na árvore.

A criação de um *i-node* envolve a alocação de um identificador livre e a inserção do registro na árvore de *i-nodes*, vinculando os metadados do arquivo às estruturas de armazenamento. A leitura realiza a busca do *i-nodes* pela chave na árvore, recuperando seus metadados diretamente do disco. A atualização é tratada como uma reescrita controlada dos campos do *i-node*, refletindo modificações em tamanho, marcações de tempo ou referências de blocos. Por fim, a exclusão remove a entrada da árvore, preservando, contudo, a referência para recuperação posterior, funcionalidade associada ao mecanismo que será descrito na Seção 4.7.

A estrutura do *i-node* também contém campos que registram os intervalos de blocos associados ao arquivo, denominados `bspans`, que definem a localização física dos dados dentro da partição. Essa estrutura, análoga aos *extents* utilizados em sistemas de arquivos como Ext4, ZFS e Btrfs, é detalhada na Seção 4.5.

Cada *i-node* do BSFS mantém duas estruturas de controle relacionadas ao endereçamento de blocos de dados. A primeira é uma lista direta de até 16 `bspans`, utilizada enquanto o arquivo pode ser representado por um número reduzido de intervalos contíguos de blocos. Quando o número de `bspans` excede o limite da lista direta, o *i-node* passa a referenciar uma árvore B de `bspans`, cuja raiz é indicada pelo campo `btree_root`. Nessa configuração, a árvore é utilizada para indexar e localizar os intervalos adicionais, permitindo que o sistema gerencie arquivos de grande porte sem perda de desempenho ou limitação de tamanho.

4.5 Endereçamento de Blocos

O BSFS emprega uma estratégia híbrida para o endereçamento de blocos de dados, combinando mapeamento direto e indexação em árvore B. A base desse mecanismo é a estrutura `bspan`, projetada especificamente para o sistema e inspirada nos *extents* utilizados em sistemas como o Ext3 e Ext4.

Cada `bspan` representa um intervalo contíguo de blocos físicos alocados para um arquivo, definido por três campos principais: o número do bloco lógico inicial (`file_blk`), o número do bloco físico correspondente (`disk_blk`) e o comprimento do intervalo (`length`). Assim, um único `bspan` pode mapear diversos blocos consecutivos, reduzindo a fragmentação e simplificando os cálculos de endereçamento durante operações de leitura e escrita.

Dentro de cada *i-node*, o BSFS mantém uma lista direta de até 16 `bspans`, utilizada enquanto o número de intervalos do arquivo permanece pequeno, conforme o exemplo da Figura 4.3. Quando o arquivo cresce e excede o limite dessa lista, o sistema inicializa uma árvore B dedicada de `bspans`. O campo `bspan_root` do *i-*

node armazena o número do bloco que contém o nó raiz dessa árvore, que é utilizada para indexar intervalos adicionais. A Figura 4.4 apresenta a estrutura completa de endereçamento de `bspans` em um dispositivo físico.

Cada nó da árvore segue o mesmo formato geral das demais árvores do BSFS, com chaves correspondendo aos blocos lógicos do arquivo e valores contendo os descritores `bspan`. Essa estrutura hierárquica garante que o acesso e a atualização de `bspans` permaneçam eficientes mesmo para arquivos grandes, com complexidade logarítmica (altura da árvore B cresce em proporção $O(\log n)$ [5]) em relação ao número total de intervalos.

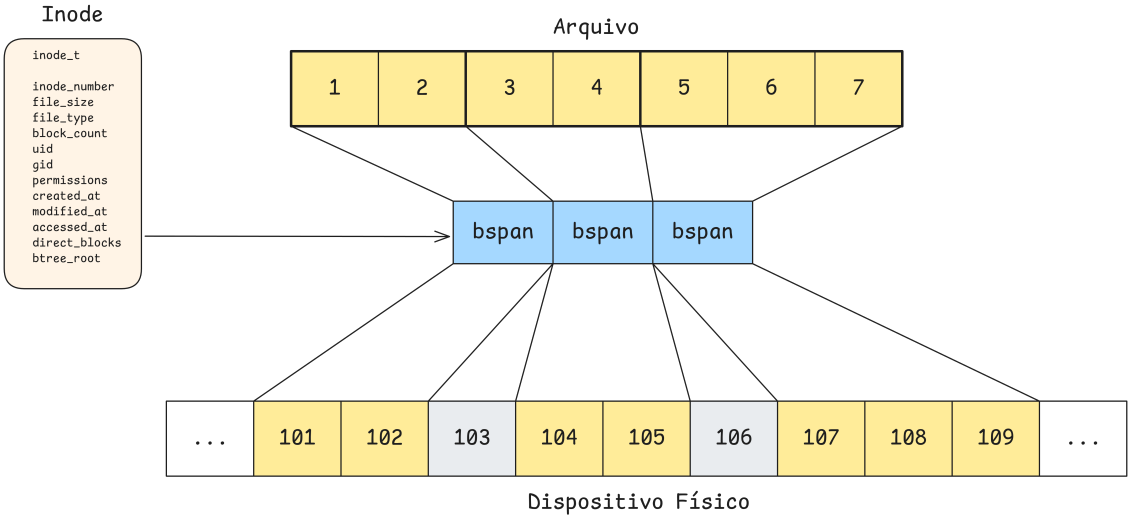


Figura 4.3: Exemplo de endereçamento direto de blocos com *bspans* implementado no BSFS.

Durante uma operação de escrita, o BSFS procura no *bitmap* o primeiro intervalo contíguo de blocos livres com comprimento suficiente para acomodar a quantidade de dados que precisam ser gravados, utilizando a política *first-fit* descrita na Seção 4.2. Caso o novo intervalo possa ser mesclado com o último `bspan` da lista ou da árvore (contiguidade física), ele é expandido. Caso contrário, é criado um novo `bspan` e inserido na posição adequada na lista direta ou, se necessário, na árvore. A leitura utiliza o caminho inverso: dado um deslocamento dentro do arquivo, o BSFS verifica inicialmente a lista direta de `bspans` para identificar o intervalo correspondente; caso o bloco solicitado não esteja coberto por nenhum dos registros da lista, a busca continua na árvore B de `bspans`, que é percorrida até localizar o intervalo que contém o bloco lógico solicitado.

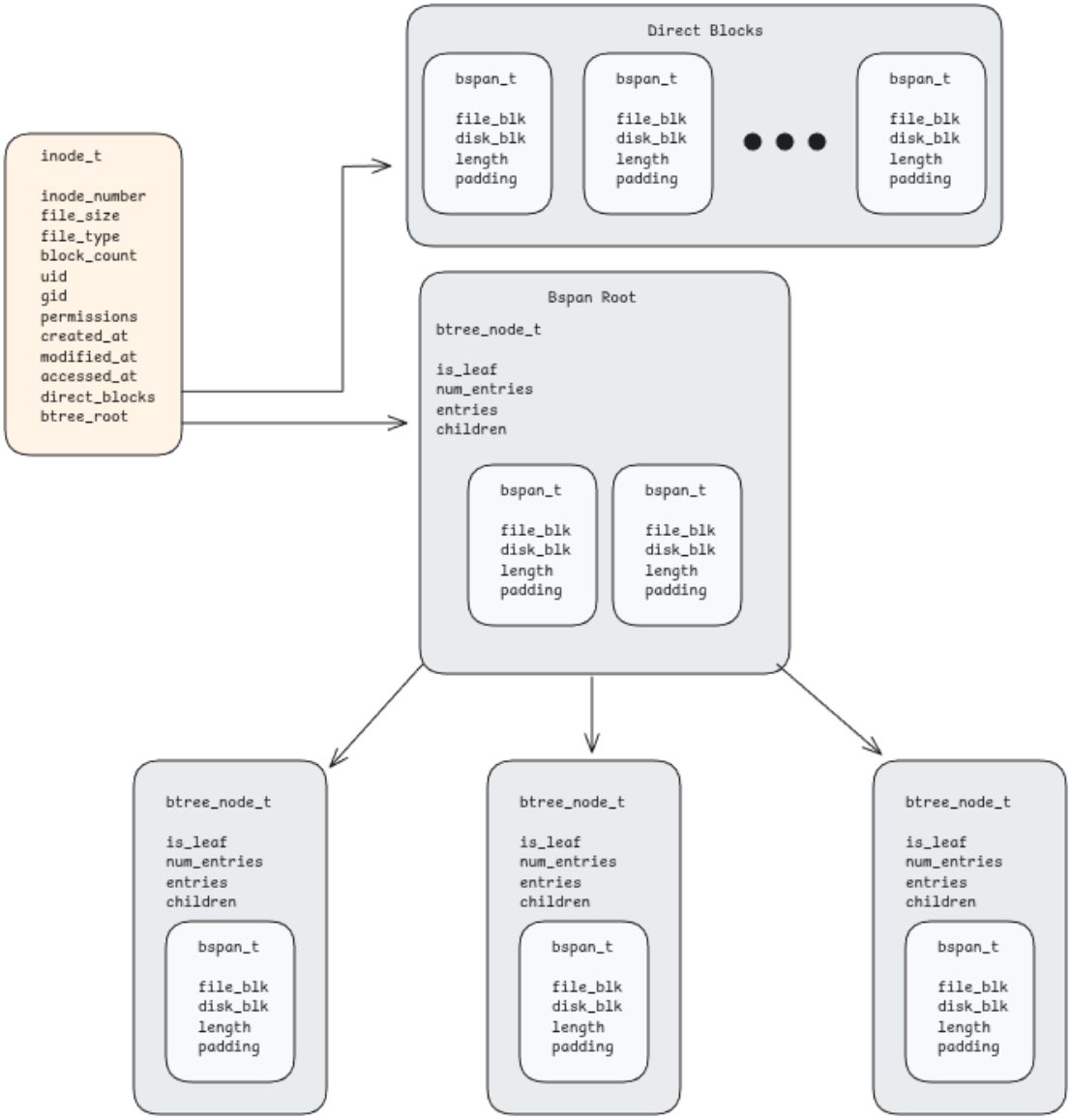


Figura 4.4: Exemplo completo de endereçamento de blocos com *bspans* implementado no BSFS.

A opção por esse modelo híbrido elimina a necessidade de blocos indiretos simples, duplos e triplos, tradicionalmente usados em sistemas como o Ext2. Além disso, a combinação de lista direta e árvore B permite equilibrar simplicidade e escalabilidade no endereçamento de blocos. No contexto do BSFS, considera-se *arquivo pequeno* aquele cujos intervalos de blocos podem ser completamente representados pela lista direta de até 16 `bspans`, sem necessidade de estruturas adicionais. Arquivos que excedem esse limite passam a utilizar, além da lista direta, uma árvore B dedicada para indexar intervalos adicionais, sendo classificados como *arquivos grandes*. Essa distinção é estrutural e não depende do tamanho lógico do arquivo, mas do

número de regiões contíguas necessárias para armazená-lo, o que reflete diretamente o grau de fragmentação da partição.

4.6 Implementação de Diretórios

A Figura 4.5 mostra como os diretórios são implementados no BSFS. Cada nó da árvore B armazena um conjunto de entradas de diretório (*directory entries*). Cada entrada associa um nome de arquivo `filename` ao seu número de *i-node* (`inode_number`), além de registrar o tipo do objeto `file_type` (arquivo comum ou diretório) e o tamanho `file_size` (em *bytes*) conhecido no momento do registro. A estrutura é de tamanho fixo e contém: o nome do arquivo com até 255 caracteres, o *hash* do nome, o *i-node* associado e os metadados mínimos para navegação (tipo e tamanho). A limitação do nome a 255 caracteres e o *layout* fixo foram adotados por simplicidade e previsibilidade de armazenamento, facilitando o acesso direto e a organização das páginas em disco.

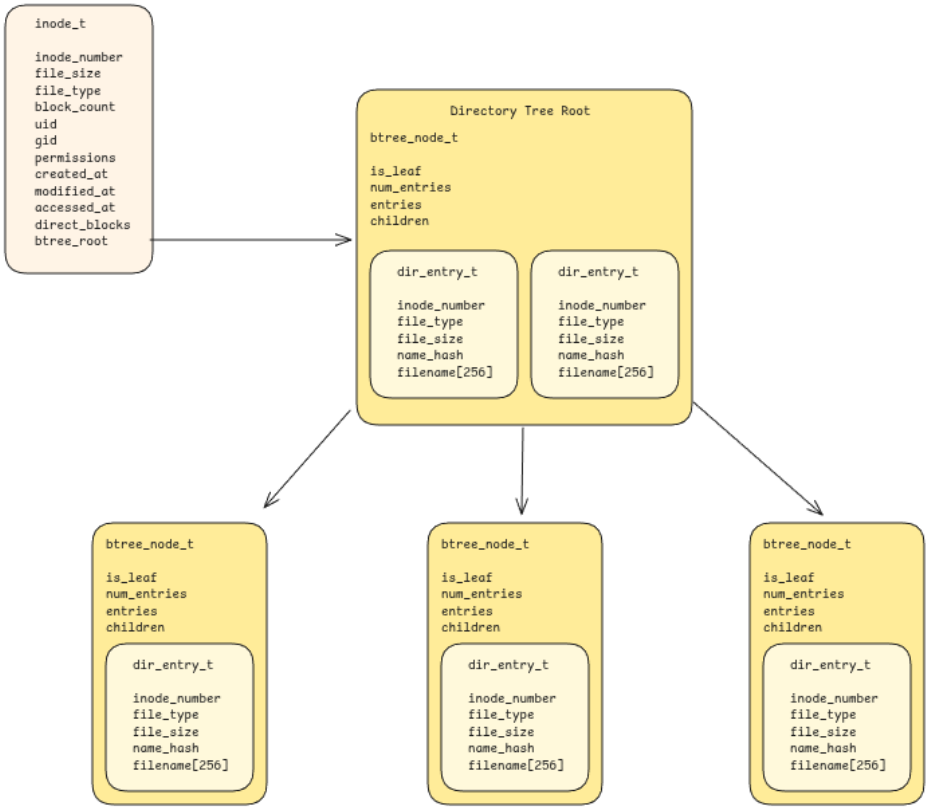


Figura 4.5: Exemplo de implementação de diretórios no BSFS.

Cada diretório é representado por um *i-node* específico, cujo campo `btree_root` não aponta para uma árvore de `bspans` (como ocorre em arquivos regulares), mas sim para a raiz da árvore B que armazena as entradas do diretório. Essa diferenciação

permite que a mesma estrutura de *i-nodes* seja utilizada de forma genérica, mudando apenas o tipo de dado indexado. Assim, arquivos e diretórios compartilham o mesmo modelo de indexação hierárquica, mas operam sobre conteúdos distintos, ou seja, `bspans` para arquivos e entradas para diretórios.

A chave de ordenação utilizada na árvore B de diretórios é o *hash* do nome, produto do algoritmo *djb2*, conforme Seção 2.2.6, escolhido por sua simplicidade e por produzir chaves de tamanho fixo, o que reduz o custo de comparação em disco e tende a distribuir as entradas uniformemente entre os nós. O nome completo permanece armazenado dentro da entrada; assim, em caso de colisão de *hash*, o sistema confirma a correspondência comparando o nome íntegro com o solicitado antes de concluir a operação.

A inserção de uma nova entrada em um diretório consiste em calcular o *hash djb2* do nome, montar a entrada fixa com nome, *hash*, *i-node*, tipo e tamanho, e inserir o registro na árvore B do diretório pai usando o *hash* como chave. A remoção elimina a chave correspondente na árvore, e inicia o processo de recuperação, que será abordado na Seção 4.7.

A busca calcula o *hash* do nome, percorre a árvore B pelo *hash* e, ao localizar o registro candidato, realiza a verificação do nome completo para resolver colisões e retornar o *i-node* correto. Dessa forma, busca, inserção e remoção mantêm complexidade logarítmica em relação ao número de entradas do diretório.

As operações sobre diretórios integram-se aos mecanismos de resolução de caminhos do BSFS. Uma sequência como `/home/usuario/documento.txt` é decomposta em componentes, e cada componente é resolvido iterativamente consultando a árvore B do diretório corrente, a partir do diretório raiz, usando o hash *djb2* do componente para localizar a entrada e, então, o *i-node* subsequente. Esse procedimento permite gerenciar caminhos longos, preservando a coerência da hierarquia e a integridade das referências entre diretórios e *i-nodes*.

4.7 Recuperação de Arquivos

A recuperação de arquivos é a principal funcionalidade adicional do BSFS, projetada para permitir a restauração de dados excluídos sem necessidade de ferramentas externas. O mecanismo é implementado por meio de uma árvore B dedicada, responsável por armazenar registros de recuperação de arquivos. Essa árvore utiliza o tipo de dados `rec_entry_t`, definido para representar cada entrada de recuperação, e é acessada a partir do bloco raiz indicado pelo campo `recovery_root` do superbloco.

A estrutura `rec_entry_t` contém os principais metadados necessários para identificar e restaurar arquivos excluídos. Cada entrada registra um identificador único (`recovery_id`), o número original do *i-node* (`inode_number`), o nome do arquivo (`original_name`), tamanho (`file_size`), permissões (`uid`, `gid`, `permissions`), *i-*

node do diretório pai (`parent_ino`), número de blocos (`block_count`), marcação de tempo (`retention_until`) e as referências aos blocos de dados originalmente associados.

A Figura 4.6 mostra os campos de entrada utilizados para recuperação em uma árvore B de exemplo. Esses campos permitem reconstruir com precisão as informações essenciais do arquivo sem depender da árvore principal de *i-nodes*. O campo de identificação exclusivo garante que múltiplos arquivos com o mesmo nome possam coexistir na árvore de recuperação sem conflito, uma vez que o acesso e a restauração são feitos com base no identificador e não no nome.

O processo de recuperação inicia-se no momento da exclusão de um arquivo. Quando o usuário remove um arquivo, alguns campos fundamentais do seu *i-node*, incluindo nome, tamanho, tipo e referências de blocos, são copiados para uma nova estrutura `rec_entry_t`. Após a cópia, o *i-node* é removido da árvore de *i-nodes*, e a nova entrada de recuperação é inserida na árvore de recuperação do sistema. Dessa forma, o registro do arquivo é mantido no sistema até que o usuário decida restaurá-lo ou que a entrada seja removida automaticamente por rotinas de limpeza.

Em situações de interrupção inesperada durante o processo de exclusão ou registro de recuperação, como a finalização abrupta do sistema operacional ou uma falha de energia, o arquivo removido pode se tornar irrecuperável. Isso ocorre porque o procedimento de cópia dos metadados para a estrutura `rec_entry_t` e sua inserção na árvore de recuperação não constituem uma operação atômica. O BSFS não implementa *journaling* ou mecanismos de transação capazes de garantir a consistência do estado do sistema em caso de falhas intermediárias. Em trabalhos futuros, a adoção de um mecanismo de *journaling* e o uso de verificações de integridade por *checksums* poderiam assegurar a atomicidade das operações de recuperação e a validação dos metadados armazenados, aumentando a confiabilidade do sistema diante de interrupções inesperadas.

Durante o período em que o arquivo permanece na árvore de recuperação, seus blocos de dados não são marcados como livres no *bitmap*, impedindo que novas alocações sobrescrevam seu conteúdo. Uma entrada permanece ativa nessa árvore enquanto estiver dentro do intervalo de retenção definido, 30 dias por padrão. Durante esse período, seus blocos continuam protegidos, exceto em situações de esgotamento do espaço livre, nas quais blocos pertencentes aos registros mais antigos podem ser reutilizados para novas alocações. Apenas quando a entrada de recuperação é definitivamente excluída, seja manualmente pelo comando `purge` ou automaticamente pelo processo de limpeza, os blocos são liberados e marcados como disponíveis no *bitmap*. Essa decisão de projeto garante integridade e segurança na restauração de arquivos, evitando perda de dados enquanto o registro de recuperação estiver ativo.

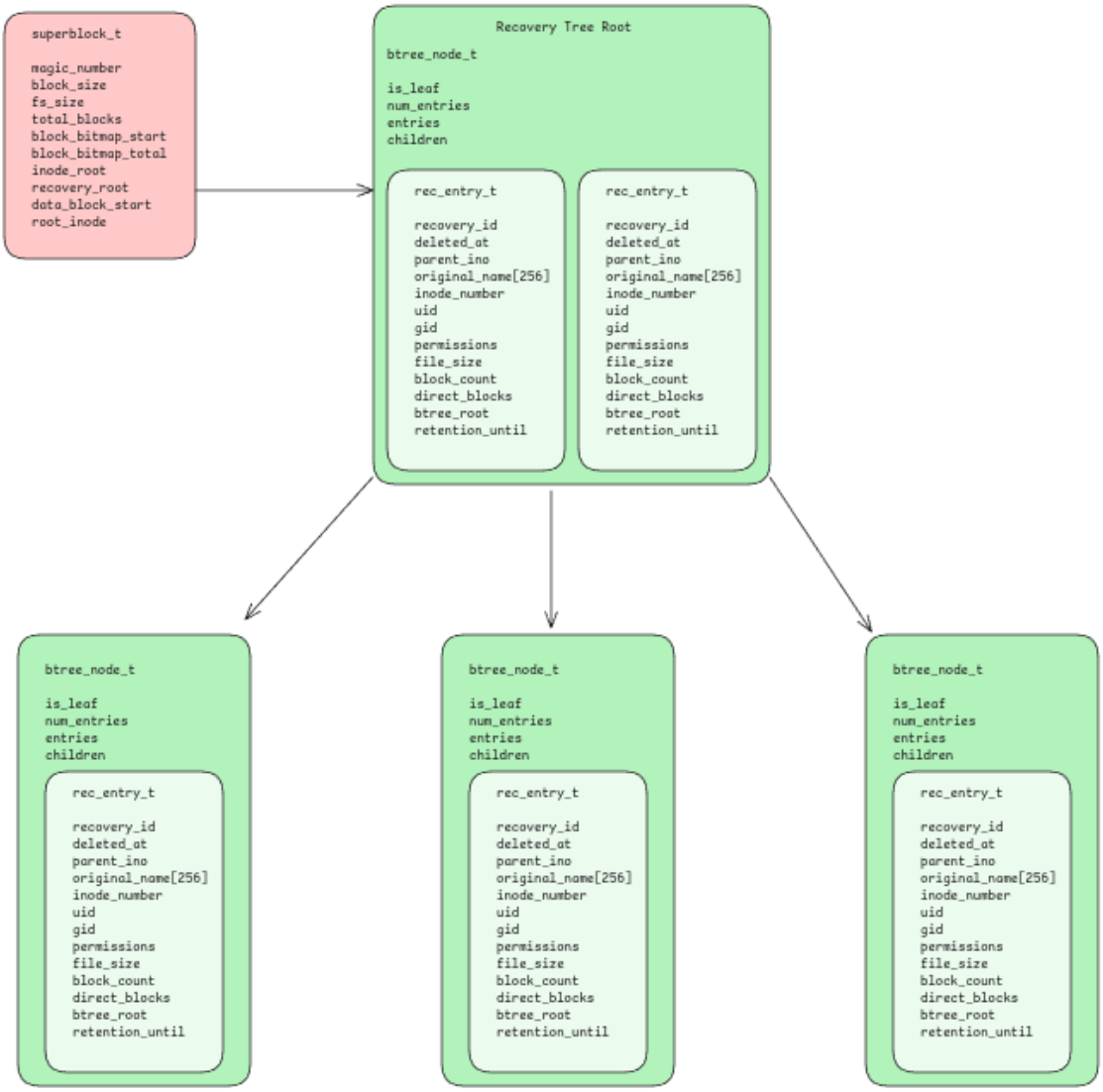


Figura 4.6: Exemplo de implementação da árvore de recuperação no BSFS.

A manutenção da árvore de recuperação inclui rotinas de limpeza periódica, responsáveis por remover registros antigos e liberar os recursos associados. O módulo de coleta de recuperação implementa um procedimento de varredura que identifica entradas cujo tempo de exclusão ultrapassa o limite estabelecido de 30 dias por padrão e remove essas entradas do sistema. Esse processo de coleta de lixo (*garbage collection*) assegura que a árvore de recuperação não cresça indefinidamente e que o espaço em disco seja reaproveitado conforme o uso.

Quando o sistema de arquivos atinge sua capacidade total, a rotina de alocação de novos blocos para escrita executa uma verificação no estado da partição. Caso não haja espaço livre suficiente para a nova operação, o BSFS aciona o módulo de coleta de recuperação para liberar blocos ocupados por arquivos mais antigos armazenados na árvore de recuperação. As entradas são removidas seguindo uma

ordem cronológica, priorizando os registros de exclusão mais antigos, de modo a preservar os arquivos mais recentes. Essa estratégia permite que o sistema continue operando mesmo em condições de saturação do espaço disponível, garantindo a continuidade das operações de escrita sem comprometer a integridade dos dados ainda recuperáveis.

O sistema oferece ainda comandos específicos para interação com o mecanismo de recuperação. Por exemplo, o comando `recovery` permite listar os arquivos disponíveis para restauração, o comando `restore` realiza a recriação do arquivo original a partir das informações armazenadas na árvore de recuperação e o comando `purge` remove permanentemente as entradas selecionadas, liberando seus blocos de dados. O funcionamento detalhado desses comandos e sua integração com a interface de usuário serão apresentados na Seção 4.8.

4.8 Interface de Interação com o Usuário

O BSFS dispõe de dois programas principais para interação em espaço de usuário: o formatador, responsável pela criação e inicialização do sistema de arquivos, e a interface *shell-like*, que permite a execução de comandos similares aos do Linux. O formatador é responsável por preparar um dispositivo de blocos para ser utilizado com o BSFS, enquanto o interpretador de comandos oferece uma interface para navegação, manipulação de arquivos e uso das funcionalidades de recuperação apresentadas na Seção 4.7. A seguir, a Seção 4.8.1 descreve em detalhes o funcionamento do formatador.

4.8.1 O Formatador (`mkfs.bsfs`)

O programa `mkfs.bsfs` é responsável pela criação e formatação de um novo sistema de arquivos BSFS em um dispositivo de blocos do Linux. Seu funcionamento é análogo ao de ferramentas tradicionais de formatação, como `mkfs.ext4` ou `mkfs.btrfs`, mas implementado em espaço de usuário.

Durante a execução, o programa recebe como argumento o caminho absoluto do dispositivo a ser formatado e procede com a criação das estruturas fundamentais do BSFS. Inicialmente, a partir do tamanho do bloco (`block_size`), o número total de blocos disponíveis é calculado considerando o tamanho total da partição. Em seguida, é criado e gravado o superbloco, que contém informações essenciais de configuração, como `magic_number`, `block_bitmap_start`, `inode_root`, `recovery_root`, e os delimitadores da área de dados, abordados na Seção 4.1.

Após a escrita do superbloco, o programa inicializa o mapa de bits responsável pelo controle de blocos livres e ocupados, conforme descrito na Seção 4.2, e grava os blocos correspondentes. Posteriormente, são criadas as árvores B vazias referentes aos *i-nodes* e à árvore de recuperação, cujos blocos raiz são referenciados no superbloco. O programa também cria o *i-node* do diretório raiz, inserindo a entrada

inicial que identifica o ponto de partida do sistema de diretórios.

4.8.2 Interpretador de Comandos (Browser)

O programa `browser` é o interpretador de comandos interativo compilado juntamente com o BSFS. Sua execução recebe como argumento o caminho para uma partição ou arquivo de blocos previamente formatado com o BSFS, permitindo ao usuário navegar, inspecionar e manipular o conteúdo do sistema de arquivos. O `browser` funciona como uma interface em linha de comando (*shell-like*), implementada em espaço de usuário, e mantém uma sessão interativa em que cada comando é lido, interpretado e executado por meio das funções disponibilizadas pela API interna.

Os comandos de manipulação de arquivos e diretórios seguem a semântica tradicional do Linux, de modo a proporcionar uma experiência familiar ao usuário. Abaixo, apresentam-se todos os comandos suportados (conforme a saída do comando `help`), organizados por categoria. A Tabela 4.1 mostra os comandos de navegação que foram implementados. Por outro lado, as Tabelas 4.2 e 4.3 mostram os comandos implementados para operações em diretórios, arquivos e permissões. Por último, a Tabela 4.4 apresenta os comandos gerais de ajuda e a Tabela 4.5 as ferramentas de recuperação e manutenção da árvore de recuperação de dados.

Tabela 4.1: Comandos de navegação

Comando	**Função**
`cd`	Altera o diretório atual.
`whoami`	Exibe o UID/GID atual e a `umask`.
`su`	Troca a identidade de sessão (teste).
`umask`	Exibe/define a `umask`.
`pwd`	Mostra o diretório de trabalho atual.
`ls`	Lista o conteúdo do diretório.
`list`	Alias para `ls`.

Tabela 4.2: Operações de diretório

Comando	**Função**
`mkdir`	Cria um novo diretório.
`rmdir`	Remove um diretório vazio.

Tabela 4.3: Operações de arquivo

Comando	**Função**
`touch`	Cria um arquivo vazio.
`rm`	Remove um arquivo (opção `rm --all [dir]` varre arquivos com medição de tempo).
`cp`	Copia um arquivo.
`mv`	Move/renomeia arquivo ou diretório.
`cat`	Exibe o conteúdo do arquivo.
`fill`	Cria muitos arquivos de determinado tamanho e reporta o tempo decorrido.
`chmod`	Altera permissões (octal, ex.: 755).
`chown`	Altera o proprietário (apenas root).
`chgrp`	Altera o grupo (apenas root).
`echo`	Exibe texto ou escreve em arquivo.
`dd`	Gera/duplica conteúdo com bloco e contagem.

Tabela 4.4: Comandos de sistema

Comando	**Função**
`info`	Exibe informações do sistema de arquivos.
`help`	Mostra o resumo de ajuda dos comandos.
`exit`	Encerra o `browser` do BSFS.

Tabela 4.5: Recuperação de arquivos

Comando	**Função**
`recovery`	Lista entradas da árvore de recuperação (filtro por nome exato ou identificador) com saída tabular.
`restore`	Restaura por número identificador ou por nome (com desambiguação). Destino opcional; senão, diretório pai original. Valida permissões; reconstrói o *i-node* a partir da cópia armazenada na entrada de recuperação.
`purge`	Limpa a árvore de recuperação: sem argumentos, remove entradas com mais de 30 dias; `purge [id]` remove uma entrada específica; `purge oldest [N]` remove as N mais antigas. Exibe contagem removida.

Os comandos de manipulação de arquivos e diretórios, como `ls`, `cd`, `mkdir`, `rmdir`, `touch`, `cp`, `mv`, `rm`, `cat` e `fill`, seguem a mesma semântica e comportamento

dos sistemas Unix tradicionais. O programa `browser` mantém uma camada de abstração entre os comandos e a API do sistema de arquivos, que realiza as chamadas de manipulação de estruturas de dados, leitura e escrita. Os comandos de sistema, como `info`, `help` e `exit`, complementam a interface, fornecendo meios de consulta, documentação e encerramento da sessão.

O BSFS utiliza um modelo de controle de acesso do tipo *Discretionary Access Control* (DAC), compatível com o esquema de permissões do Linux. Cada *i-node* contém campos de UID, GID e modo de acesso, avaliados durante as operações de leitura e escrita. Comandos como `su`, `chmod`, `chown` e `chgrp` reproduzem as mesmas funcionalidades do ambiente Unix, permitindo a simulação de diferentes usuários, alteração de permissões e gerenciamento de grupos, o que torna o comportamento do sistema de arquivos compatível com o modelo de segurança de arquivos.

O comando `recovery` tem como objetivo listar as entradas armazenadas na árvore de recuperação, conforme descrito na Seção 4.7. Sem argumentos, percorre toda a árvore em ordem e exibe uma tabela com identificador, tipo, tamanho, horários de exclusão, período de retenção, *i-node* do diretório pai e nome original do arquivo. Quando recebe um argumento de nome, o comando realiza um filtro exato, exibindo apenas as entradas correspondentes, o que facilita a localização de arquivos específicos.

O comando `restore` permite reconstruir arquivos previamente excluídos, utilizando os dados armazenados nas entradas de recuperação. A restauração pode ser solicitada pelo identificador único (`recovery_id`) ou pelo nome do arquivo. Quando há múltiplas entradas com o mesmo nome, o programa apresenta uma lista para desambiguação. O diretório de destino pode ser especificado como segundo argumento; caso contrário, o arquivo é restaurado no diretório original ou no diretório raiz do sistema de arquivos. O processo inclui a verificação das permissões do destino e a reconstrução completa do *i-node* a partir de sua cópia armazenada na entrada de recuperação.

O comando `purge` é utilizado para remover definitivamente entradas da árvore de recuperação, liberando seus blocos no *bitmap*. Quando executado sem parâmetros, ele remove automaticamente arquivos cuja exclusão tenha ocorrido há mais de 30 dias. A opção `purge [ID]` remove uma entrada específica pelo seu `recovery_id`, e `purge oldest [N]` exclui as `N` entradas mais antigas. Após a execução, o comando informa a quantidade de registros removidos, garantindo o reaproveitamento do espaço em disco e a manutenção periódica da estrutura de recuperação.

Os comandos apresentados nesta seção descrevem a interface de interação oferecida pelo BSFS e suas principais funcionalidades. Exemplos detalhados de uso, incluindo sequências de execução e saídas produzidas pelo sistema, encontram-se no Apêndice A, que complementa esta seção com demonstrações práticas.

4.9 Resultados de Desempenho

Esta seção apresenta os resultados obtidos nos testes de desempenho realizados com o BSFS e outros sistemas de arquivos abordados no Capítulo 3: Ext4, Btrfs e ZFS. A Tabela 4.6 apresenta as configurações do *hardware* utilizado nos testes.

Tabela 4.6: Especificações de *hardware* da estação de testes.

Componente	**Descrição**
Processador	AMD Ryzen 7 5700G with Radeon Graphics
Caches	512 KiB L1, 4 MiB L2 e 16 MiB L3
Placa-mãe	ASUS A520M K V2
Chipset	AMD A520 (Renoir/Cezanne Platform)
Memória RAM	16 GB DDR4 (2×8 GB, 2133 MHz, dual channel)
Controladora SATA	AMD 500 Series Chipset SATA Controller
Armazenamento do sistema	SSD NVMe Kingston SNV3S500G (500 GB) — utilizado exclusivamente para o sistema operacional
Armazenamento de testes	SSD SATA III ADATA SU630 (447 GB) — dedicado aos benchmarks de sistemas de arquivos

Os testes foram realizados na distribuição *Arch Linux*, utilizando o conjunto de ferramentas e versões de software mostrados nas Tabelas 4.7 e 4.8.

Tabela 4.7: Especificações de *software* do ambiente de testes.

Componente	**Versão / Descrição**
Kernel	Linux 6.12.51-1-lts #1 SMP PREEMPT_DYNAMIC x86_64 GNU/Linux
Compilador	GCC 15.2.1 (2025-08-13)

Tabela 4.8: Programas de formatação utilizados nos testes.

Sistema de Arquivos	**Ferramenta / Versão**
BSFS	`mkfs.bsfs v1.0`
Ext4	`mke2fs 1.47.3 (8-Jul-2025)`
Btrfs	`btrfs-progs v6.17`
ZFS	`zfs-2.3.4-1`

4.9.1 Metodologia de Teste

Para a avaliação de desempenho foram desenvolvidos roteiros de teste executados diretamente sobre partições de disco formatadas (`/dev/sdX`) com cada um dos se-

guintes sistemas de arquivos: BSFS, Ext4, ZFS e Btrfs. Ambos os procedimentos utilizam os mesmos conjuntos de arquivos previamente gerados a partir de `/dev/urandom`: 100 arquivos de 10 MiB e um arquivo único de 1 GiB, preservados e reutilizados em todas as execuções. Embora o dispositivo `/dev/urandom` produza dados não determinísticos, o uso de um conjunto fixo de arquivos assegura que todos os sistemas de arquivos sejam submetidos à mesma carga de escrita, garantindo reprodutibilidade dos testes e comparabilidade entre os resultados.

Em ambos os experimentos, cada ciclo consiste na gravação e remoção dos 100 arquivos de 10 MiB com conteúdo aleatório, na mesma ordem, 10 vezes. Em seguida, ocorre a gravação e remoção do arquivo único de 1 GiB, também por 10 vezes. No primeiro experimento, os testes no BSFS foram executados no *browser* em sessão única, alimentados por comandos redirecionados de um arquivo de texto. Esse procedimento permitiu reproduzir interações com o sistema de arquivos sem intervenção manual.

O primeiro experimento foi conduzido em duas variações: a primeira executa apenas a remoção dos arquivos após cada ciclo, preservando as entradas de recuperação na árvore; a segunda inclui a execução do comando de limpeza `purge`, que remove os registros mais antigos da árvore de recuperação e libera os blocos associados. Essa distinção possibilita avaliar o impacto direto do mecanismo de recuperação sobre o desempenho do sistema. O sistema de arquivos registra o tempo de execução de cada operação de gravação por meio de um relógio monotônico interno.

O segundo experimento foi projeto para medir o desempenho dos sistemas de arquivos Ext4, Btrfs e ZFS. Nesse caso, cada sistema de arquivos é avaliado separadamente sob as mesmas condições de carga e parâmetros de teste. Antes de cada rodada, os arquivos anteriores são removidos para evitar reutilização de dados e interferência nos tempos subsequentes. A medição de tempo baseia-se na captura de *timestamps* em nanosegundos via `date +%s%N` e, entre as repetições, o cache de páginas do *kernel* é limpo para minimizar o impacto de armazenamento em memória.

4.9.2 Resultados e Discussão

Nesta subseção, são apresentados os resultados de tempo de gravação de arquivos para duas cargas: 100×10 MiB e 1×1 GiB. As Figuras 4.7 e 4.8 exibem, para cada sistema de arquivos (BSFS nas variantes com e sem `purge`, além de Ext4, Btrfs e ZFS), barras horizontais com o tempo médio (em segundos) e, em cada barra, os valores de máximo e mínimo observados ao longo de 10 repetições. Nesse contexto, menor é melhor.

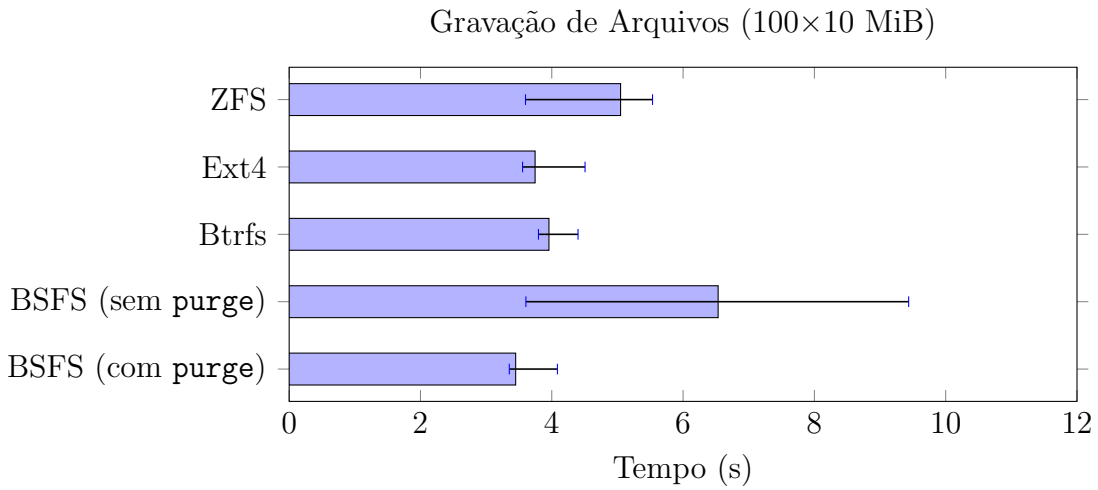


Figura 4.7: O gráfico apresenta a média de dez execuções na gravação de 100 arquivos de 10 MiB em cada um dos sistemas de arquivos.

No cenário com `purge`, o BSFS apresenta tempos médios estáveis e desempenho competitivo frente aos sistemas em espaço de *kernel*. Essa vantagem relativa decorre do menor volume de metadados atualizados a cada escrita e da ausência de métodos que garantem a integridade como *journaling* ou COW, mecanismos que, embora elevem a confiabilidade, introduzem sobrecarga adicional no processo para estes sistemas de arquivos.

Quando o `purge` não é realizado entre cada teste, cada `rm` apenas move o arquivo para a árvore de recuperação, preservando metadados e blocos de dados alocados. Assim, a cada ciclo de testes, o volume de entradas de recuperação cresce linearmente e os blocos permanecem indisponíveis ao alocador.

Nesse contexto, a política *first-fit* adotada pelo BSFS se mostra ineficiente por efetuar varreduras lineares no *bitmap* em busca do primeiro intervalo contíguo com tamanho suficiente, tendo desempenho impactado com a fragmentação crescente da partição. Os segmentos livres tornam-se menores e mais esparsos, prolongando as buscas e degradando a localidade. O efeito acumulado manifesta-se em dois pontos: no alocador de blocos, cujas varreduras ficam progressivamente mais longas à medida que o espaço contíguo diminui; e na coleta emergencial de espaço, que precisa percorrer a árvore de recuperação para liberar intervalos (`bspans`) quando a pressão por blocos aumenta. O custo dominante, portanto, está na alocação e na coleta, não na travessia das árvores B.

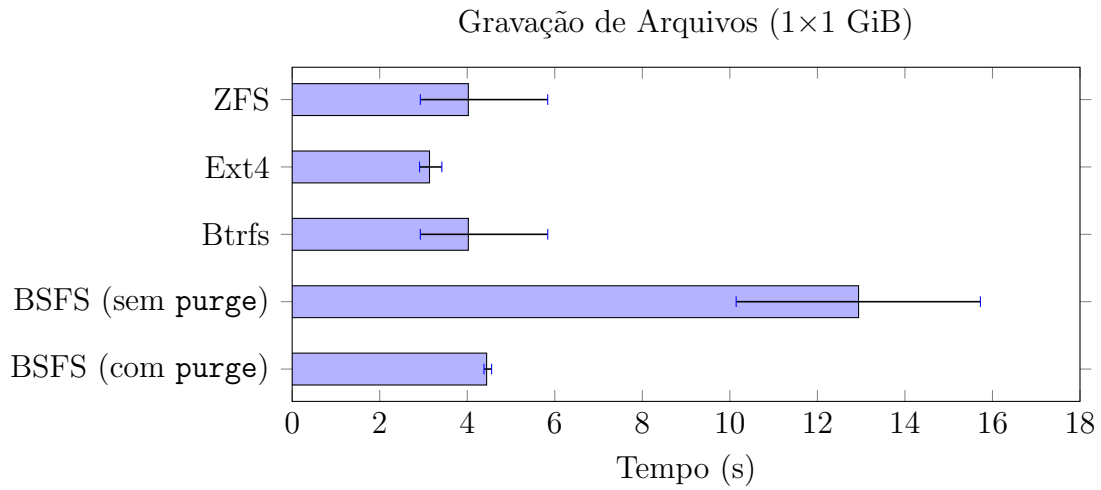


Figura 4.8: O gráfico apresenta a média de dez execuções na gravação de um arquivo de 1 GiB em cada um dos sistemas de arquivos.

A manutenção das árvores B para diretórios, *i-nodes* e `bspans` introduz custos logarítmicos para busca e atualização, os quais permanecem secundários diante do trabalho extra do alocador sob fragmentação e da necessidade de liberar espaço ao atingir alta ocupação. Assim, o aumento do tempo ao longo das rodadas não decorre de lentidão estrutural das árvores, mas do comportamento intencional do BSFS em reter versões na recuperação quando o `purge` não é aplicado, somado à sensibilidade do *first-fit* à fragmentação.

Apesar de não ter se mostrado o principal fator responsável pela diferença de desempenho, a execução do BSFS em espaço de usuário exerce influência nos resultados. Enquanto sistemas como Ext4, Btrfs e ZFS operam em modo *kernel*, o BSFS depende de chamadas de sistema para interagir com o *hardware* e com o próprio núcleo do sistema operacional. Cada operação de leitura ou escrita em disco implica transições entre os modos de usuário e de *kernel*, trocas de contexto e verificações de proteção de memória. Esses procedimentos acrescentam um *overhead* adicional, que não está presente nas implementações nativas em espaço de *kernel*. Love [9] discorre que cada chamada de sistema envolve uma alternância de contexto entre esses dois modos de execução, o que resulta em um custo mensurável de tempo mesmo em sistemas otimizados como o Linux.

Para trabalhos futuros, é necessário planejar um novo mecanismo de gerenciamento de blocos livres, buscando uma abordagem mais eficiente e integrada à lógica de recuperação de arquivos. A atual estratégia de varredura sequencial (*first-fit*) mostrou-se funcional, mas pouco adaptável a cenários de fragmentação e retenção prolongada de dados. Soluções alternativas, como alocadores baseados em agrupamento, ou estruturas hierárquicas de *bitmap*, podem melhorar a previsibilidade e reduzir o custo das operações de alocação. Além disso, uma readequação do projeto para execução em espaço de *kernel* é desejável, de modo a eliminar o *overhead* as-

sociado às chamadas de sistema e permitir integração direta com as rotinas nativas de entrada e saída (I/O) do Linux, possibilitando uma avaliação de desempenho comparável aos outros sistemas de arquivos.

4.10 Sumário

Este capítulo apresentou a implementação do *Basic Solution File System* (BSFS), um sistema de arquivos desenvolvido em linguagem C e executado inteiramente em espaço de usuário no ambiente Linux. Foram descritas suas principais estruturas internas: o superbloco, responsável pela configuração e localização das demais áreas do sistema; o gerenciamento de blocos livres baseado em *bitmaps* encadeados; e o uso de árvores B como estrutura para indexação de *i-nodes*, blocos de dados e diretórios.

Destacou-se o modelo híbrido de endereçamento, que combina listas diretas de `bspans` para arquivos pequenos e árvores B dedicadas para arquivos maiores, bem como o mecanismo de recuperação de arquivos excluídos, implementado por meio de estruturas `rec_entry_t` armazenadas em uma árvore de recuperação independente. Também foram detalhados os utilitários do sistema, incluindo o formatador `mkfs.bsfs` e o interpretador interativo `browser`, que oferece uma interface semelhante a um *shell* tradicional, com comandos de manipulação de arquivos e diretórios, além das operações específicas de recuperação, restauração e remoção de registros.

Os testes de desempenho compararam o BSFS a sistemas de arquivos utilizados com o Linux, como Ext4, Btrfs e ZFS, em diferentes cenários de escrita sequencial. Os resultados mostraram que o BSFS apresenta tempos de escrita maiores quando a rotina de `purge` não é executada, devido ao acúmulo de metadados na árvore de recuperação e à fragmentação crescente do espaço livre. Identificou-se que o uso do algoritmo *first-fit* para alocação de blocos e a ausência de estratégias de agrupamento agravam o tempo de busca em situações de alta ocupação. Por outro lado, quando o `purge` é aplicado entre as execuções, o desempenho torna-se estável e próximo ao dos sistemas de arquivos de referência.

Também foi discutido o impacto do BSFS operar em espaço de usuário, o que implica um *overhead* adicional decorrente das transições entre os modos de execução e das chamadas de sistema necessárias para interação com o *kernel*. Apesar disso, o sistema de arquivos se mostrou funcional, consistente e capaz de recuperar arquivos excluídos sem perda de integridade.

Capítulo 5

Conclusão

Art is never finished, only abandoned.

– Leonardo da Vinci

Este capítulo apresenta as considerações finais a respeito do desenvolvimento e avaliação do *Basic Solution File System* (BSFS), consolidando os resultados obtidos e destacando as principais limitações e possibilidades de evolução do projeto.

O Capítulo 1 introduziu o contexto e a motivação para o estudo, ressaltando a importância dos sistemas de arquivos na organização e persistência de dados em sistemas operacionais do tipo Unix. O Capítulo 2 abordou os fundamentos teóricos necessários à implementação de um sistema de arquivos, discutindo conceitos como *i-nodes*, diretórios, gerenciamento de espaço livre com *bitmaps* e o impacto do tamanho de blocos no desempenho. O Capítulo 3 apresentou os principais sistemas de arquivos utilizados em ambientes Linux: Ext2, Ext3, Ext4, ZFS e Btrfs, destacando suas arquiteturas, os mecanismos de gerenciamento de metadados e as estratégias empregadas para organização, alocação e preservação de integridade. Por fim, o Capítulo 4 descreveu detalhadamente a implementação do BSFS, suas estruturas de dados e os resultados comparativos de desempenho em relação aos sistemas de arquivos abordados anteriormente.

O BSFS atingiu o objetivo proposto de implementar um sistema de arquivos funcional em espaço de usuário, projetado para operar sobre dispositivos de blocos no Linux. Sua principal colaboração foi um mecanismo para recuperação nativa de arquivos excluídos, ausente nos sistemas convencionais, que mantém registros estruturados de metadados e blocos de dados para restauração futura. Embora o sistema não tenha apresentado desempenho competitivo quando a lógica de recuperação é aplicada em comparação a sistemas de arquivos otimizados como Ext4, ZFS e Btrfs, o BSFS cumpriu seu papel como experimento prático e prova de conceito, demonstrando a viabilidade de uma implementação em espaço de usuário.

As principais limitações observadas no BSFS estão relacionadas à sua arquitetura e às decisões de projeto adotadas. O mecanismo de recuperação, embora funcional, tende a aumentar o volume de metadados mantidos em disco, especialmente quando a rotina de limpeza (`purge`) não é executada com frequência, o que leva ao crescimento contínuo da árvore de recuperação e à ocupação prolongada de blocos. Em conjunto, a política de alocação baseada em *first-fit* mostrou-se sensível à fragmentação, uma vez que realiza varreduras lineares no *bitmap* até localizar o primeiro intervalo contíguo livre, tornando-se progressivamente menos eficiente à medida que o espaço disponível se fragmenta. Essa combinação faz com que o custo de alocação e coleta de blocos se torne o principal gargalo de desempenho do sistema.

Além dessas limitações, o BSFS opera integralmente em espaço de usuário, o que impõe uma sobrecarga adicional decorrente das transições entre os modos de execução e das chamadas de sistema necessárias para acesso ao disco. O BSFS ainda carece de recursos avançados presentes em sistemas de arquivos modernos, como mecanismos de concorrência e paralelismo, estratégias de alocação por localidade e suporte a *journaling*, que poderiam ampliar sua confiabilidade e eficiência em cenários reais de uso.

5.1 Trabalhos Futuros

Como trabalhos futuros, propõe-se a adaptação do BSFS para operação em modo *kernel*, por meio da implementação de um módulo dedicado ao Linux. Essa implementação permitiria eliminar a sobrecarga associada às chamadas de sistema e possibilitaria integração direta com as rotinas nativas de entrada e saída (*I/O*), aproximando o desempenho do sistema de arquivos ao de implementações como Ext4, ZFS e Btrfs.

Outra linha de aprimoramento envolve o redesenho do mecanismo de gerenciamento de blocos livres. A estratégia atual, baseada em varredura sequencial (*first-fit*), mostrou-se simples e funcional, mas pouco eficiente em cenários de alta fragmentação e retenção prolongada de dados. O desenvolvimento de um alocador mais sofisticado, possivelmente baseado em agrupamento de blocos, em estruturas hierárquicas de *bitmap* ou em algoritmos de alocação que consideram a localidade de blocos de um mesmo arquivo, poderia reduzir o custo das operações e melhorar o desempenho.

Além disso, a introdução de suporte à concorrência e paralelismo nas rotinas de leitura e escrita constitui um passo esperado na evolução do sistema, de modo a aproveitar arquiteturas *multicore* modernas e permitir processamento simultâneo de diferentes regiões das árvores de metadados.

Por fim, a implementação de mecanismos de preservação de consistência, como *journaling* ou políticas baseadas em *Copy-on-Write* (COW), representa um passo relevante para ampliar a confiabilidade do BSFS. Um módulo de *journaling* permitiria registrar transações de metadados e operações críticas, possibilitando a recuperação

automática após falhas inesperadas. Alternativamente, a adoção de uma política COW evitaria a sobrescrita direta de dados e metadados, preservando versões anteriores até a finalização das atualizações. Em ambos os casos, a combinação com verificações de integridade baseadas em *checksums* permitiria que o BSFS evoluísse para uma plataforma experimental mais robusta para estudos de tolerância a falhas e recuperação em sistemas de arquivos.

5.2 Considerações Finais

O desenvolvimento do BSFS permitiu compreender de maneira prática os princípios de funcionamento interno de sistemas de arquivos, consolidando o aprendizado teórico sobre organização, persistência e gerenciamento de dados em sistemas operacionais. Mais do que um sistema de arquivos funcional, o BSFS representa um exercício de engenharia de software de baixo nível, cujo valor reside na aplicação dos conceitos fundamentais de arquitetura de sistemas, estruturas de dados e controle de armazenamento em um projeto completo e experimental. O código fonte [21] está disponível online sob a licença GPLv3.

Referências Bibliográficas

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.10 edition, November 2023.

[2] D. J. Bernstein. djb2 Hash Function. `https://cr.yp.to`, 1991. [Online; acessado em 08-11-2025].

[3] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Sebastopol, CA, 2 edition, 1994.

[4] J. Chen, J. Wang, Z. Tan, and C. Xie. Effects of Recursive Update in Copy-on-Write File Systems: A BTRFS Case Study. *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10, 2014.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Algoritmos - Teoria e Prática*. Elsevier Editora Ltda., 3rd edition, 2012.

[6] B. Djordjevic and V. Timcenko. Ext4 File System in Linux Environment: Features and Performance Analysis. *International Journal of Computers*, 6(1):37–45, 2012.

[7] H. Goyal and C. Zoulas. Addition of Ext4 Extent and Ext3 HTree DIR Read-Only Support in NetBSD. *AsiaBSDCon*, 2017.

[8] I. Hwang, S. Kim, H. Eom, and Y. Son. Sequentialized Virtual File System: A Virtual File System Enabling Address Sequentialization for Flash-Based Solid State Drives. *Computers*, 13(11):284, 2024.

[9] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, Upper Saddle River, NJ, 3rd edition, 2010.

[10] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The New Ext4 Filesystem: Current Status and Future Plans. *Proceedings of the Linux symposium*, 2:21–33, 2007.

[11] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[12] K. Munegowda, G. T. Raju, and V. Raju. Evaluation of File Systems for Solid State Drives. *Procedings on the Second International Conference on Emerging Research in Computing, Information, Communication and Applications*, 08 2014.

[13] S. D. Pate. *UNIX Filesystems: Evolution, Design and Implementation*. John Wiley & Sons, 1999.

[14] D. M. Ritchie. The Evolution of the UNIX Time-sharing System. *AT&T Bell Laboratories Technical Journal*, 63(8):1577–1593, 1984.

[15] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage*, 9(3):9:1–9:32, 2013.

[16] R. Shinde, V. Patil, A. Bhargava, A. Phatak, and A. More. Inline Block Level Data De-duplication Technique for EXT4 File System. *Advances in Intelligent Systems and Computing*, 249, 01 2014.

[17] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, Hoboken, NJ, USA, 10th edition, 2018.

[18] A. S. Tanenbaum. *Sistemas Operacionais Modernos*. Pearson Education, São Paulo, 4 edition, 2016.

[19] A. S. Tanenbaum, J. N. Herder, and H. Bos. File Size Distribution on UNIX Systems: Then and Now. *ACM SIGOPS Operating Systems Review*, 40:100–104, Jan. 2006.

[20] N. Tehrany, K. Doekemeijer, and A. Trivedi. A Survey on the Integration of NAND Flash Storage in the Design of File Systems and the Host Storage Software Stack. *arXiv pre-print*, 2023. arXiv:2307.11866.

[21] V. Benites, G. L. Flores, F. B. Rodrigues and, Brivaldo A. da Silva Jr. Basic Solution FileSystem. `https://github.com/WorksFacom/bsfs`, 2025. [Online; acessado em 11-11-2025].

[22] W. Vogels. File System Usage in Windows NT 4.0. *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, 1999. [Online; acessado em 10-11-2025].

[23] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. *FAST '10: 8th USENIX Conference on File and Storage Technologies*, pages 29–42, 2010.

Apêndice A

Exemplo de Uso do BSFS

Neste apêndice será mostrado um exemplo de formatação de dispositivo utilizando o BSFS, criando arquivos e restaurando um arquivo removido acidentalmente pelo usuário.

A.1 Formatando um dispositivo com o BSFS

Após compilar o projeto, você pode formatar uma imagem baseada em arquivo (ou até mesmo um dispositivo de bloco) e explorá-la por meio do navegador (`browser`) do BSFS.

O primeiro passo é criar uma imagem cujo tamanho é múltiplo de 4096 bytes. No exemplo a seguir será criada uma imagem de 8 MiB:

```
$ dd if=/dev/zero of=bsfs.img bs=1M count=8
8+0 registos entrados
8+0 registos saídos
8388608 bytes (8,4 MB, 8,0 MiB) copiados, 0,00399782 s, 2,1 GB/s
```

O próximo passo é formatar o arquivo criado:

```
$ ./bin/mkfs.bsfs bsfs.img
BSFS formatted successfully
Partition size (bytes): 8388608
Block size (bytes): 4096
Total blocks: 2048
Block bitmap (start,total): 1,1
Data block start: 2
Inode root block: 2
Root inode: 1
```

É importante destacar que é possível formatar um dispositivo de blocos como, por exemplo, uma partição de disco `/dev/sda1`. O formatador irá gravar o superbloco, inicializar o mapa de *bits* do bloco, a árvores B de *i-nodes* e criar as entradas do diretório raiz.

A.2 Criando arquivos

Agora serão criados quatro arquivos de maneiras diferentes utilizando o `browser` do BSFS. O primeiro passo é acessar o sistema de arquivos recém criado com o `browser`:

```
$ ./bin/browser bsfs.img
BSFS Browser
Partition: bsfs.img
Type "help" for available commands or "info" for filesystem details
BSFS:/>
```

O `browser` suporta diversos comandos similares aos do Linux. Abaixo o comando `help`:

```
BSFS:/> help
BSFS Browser Commands:
======================
Navigation:
  cd <directory>     - Change current directory
  whoami             - Show current uid/gid and umask
  su <uid> [gid]     - Switch session identity (testing)
  umask [octal]      - Show/set umask
  pwd                - Print current working directory
  ls [directory]     - List directory contents
  list [directory]   - List directory contents (alias for ls)

Directory Operations:
  mkdir <directory>  - Create a new directory
  rmdir <directory>  - Remove an empty directory

File Operations:
  touch <filename>   - Create an empty file
  rm <filename>      - Remove a file (rm --all [dir] sweeps files with timing)
  cp <src> <dest>    - Copy a file
  mv <src> <dest>    - Move/rename a file or directory
  cat <filename>     - Display file contents
  fill <count> <size> [prefix] [bs=<n>] [pattern=index|zero|random]
       Create many files and print elapsed time (size/bs accept K/M/G)
  chmod <mode> <path>- Change permissions (octal, e.g., 755)
  chown <uid> <path> - Change file owner (root only)
  chgrp <gid> <path> - Change file group (root only)
  echo <text>        - Display text or write to file
    echo "text" > file   - Write text to file (overwrite)
    echo "text" >> file  - Append text to file
  dd if=<src> of=<dest> bs=<size> count=<blocks> - Create files of specific size
    Example: dd if=/dev/zero of=testfile bs=1M count=10 (creates 10MB file)

System:
  info               - Display filesystem information

Recovery:
  recovery [name]    - List recovery entries (optionally filter by name)
  restore <id|name> [dir] - Restore a deleted file by recovery id or by original name
                       (optional dest dir)
  purge [id|oldest N|expired] - Purge recovery entries (default: older than ~30 days)
  help               - Display this help message
  exit               - Exit the BSFS browser
```

Agora, serão criados os quatro arquivos:

```
BSFS:/> touch file.txt
File created: file.txt

BSFS:/> dd if=/dev/zero of=block.img bs=1M count=1
dd: copying 1 blocks of 1048576 bytes each...
dd: 1 blocks (1048576 bytes) copied
dd: completed in 3.978 ms (251.41 MiB/s)

BSFS:/> echo "Prof. Brivaldo" > name.txt
Text written to file: name.txt

BSFS:/> echo "BSFS FileSystem" > othername.txt
Text written to file: othername.txt
```

Foram criados os arquivos: `file.txt` vazio, `block.img` de 1MiB de dados do `/dev/zero`, `name.txt` contendo o texto `"Prof. Brivaldo"` e `othername.txt` contendo o texto `"BSFS FileSystem"`.

No próximo passo, é verificada a existência dos arquivos usando o comando de listagem `ls` e o conteúdo de um dos arquivos, para garantir que os dados estão corretos:

```
BSFS:/> ls
Listing directory: .
Name                    Type
----                    ----
.                       DIR
..                      DIR
name.txt                FILE
file.txt                FILE
othername.txt           FILE
block.img               FILE

BSFS:/> cat name.txt
"Prof. Brivaldo"
```

Nesta etapa, o arquivo `othername.txt` será removido:

```
BSFS:/> rm othername.txt
File removed: othername.txt

BSFS:/> ls
Listing directory: .
Name                    Type
----                    ----
.                       DIR
..                      DIR
name.txt                FILE
file.txt                FILE
block.img               FILE
```

O próximo passo é recuperar um arquivo removido. Notem que o arquivo não aparece mais no sistema de arquivos ao executar o comando `ls`.

A.3 Recuperando um arquivo removido

E, com o comando `restore`, o arquivo `othername.txt` será recuperado:

```
BSFS:/> recovery
Recovery entries:
ID     Type  Size     Deleted At          Retain Until        ParentIno  Name
--     ----  ----     ------------------- ------------------- ---------- ----
1      FILE  18       2025-11-13 08:44:08 2025-12-13 08:44:08 1          othername.txt
```

Ao se digitar o comando `recovery`, a estrutura de dados do BSFS de recuperação é acessada para verificar os arquivos que foram removidos, mostrando suas informações como o nome do arquivo, quando foi removido e até quando sua retenção é esperada. O próximo passo é recuperá-lo com o comando `restore`:

```
BSFS:/> restore othername.txt
Restored 'othername.txt' (id 1)

BSFS:/> ls
Listing directory: .
Name                      Type
----                      ----
.                         DIR
..                        DIR
name.txt                  FILE
file.txt                  FILE
othername.txt             FILE
block.img                 FILE

BSFS:/> cat othername.txt
"BSFS FileSystem"
```

No exemplo, o arquivo foi restaurado pelo seu nome (poderia ter sido restaurado pelo ID na tabela de recuperação). Como observado, o arquivo foi restaurado e voltou a aparecer no sistema de arquivos ao executar o `ls` e o seu conteúdo continuou intacto. Com isso, está finalizada a demonstração de uso, formatação e recuperação do BSFS.