Rafael Breno Rocha Reis

UM SISTEMA DE ARQUIVO PARA DISPOSITIVOS DE ARMAZENAMENTO COM ACESSO ALEATÓRIO

Formiga - MG

2019

Rafael Breno Rocha Reis

UM SISTEMA DE ARQUIVO PARA DISPOSITIVOS DE ARMAZENAMENTO COM ACESSO ALEATÓRIO

Monografia do trabalho de conclusão de curso apresentado ao Instituto Federal Minas Gerais - Campus Formiga, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais

Campus Formiga

Bacharelado em Ciência da Computação

Orientador: Prof. M.e Everthon Valadão

Formiga - MG

2019

Reis, Rafael Breno Rocha.
004 Um sistema de arquivo para dispositivos de armazenamento com Acesso aleatório / Rafael Breno Rocha Reis. -- Formiga : IFMG, 2019.
99p.. : il.

Orientador: Prof. Msc. Everthon Valadão
Trabalho de Conclusão de Curso – Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais – *Campus* Formiga.

1. Sistema de arquivos. 2. Análise de desempenho. 3. Dispositivos de armazenamento. 4. Acesso aleatório. I. Título.

CDD 004

Ficha catalográfica elaborada pela Bibliotecária Msc. Naliana Dias Leandro CRB6-1347

Rafael Breno Rocha Reis

UM SISTEMA DE ARQUIVO PARA DISPOSITIVOS DE ARMAZENAMENTO COM ACESSO ALEATÓRIO

Monografia do trabalho de conclusão de curso apresentado ao Instituto Federal Minas Gerais - Campus Formiga, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em: 15 de junho de 2019.

BANCA EXAMINADORA

M.e Everthon Valadão
Professor no IFMG *Campus* Formiga
(Orientador)

M.e Diego Mello da Silva
Professor no IFMG *Campus* Formiga

M.e Walace de Almeida Rodrigues
Professor no IFMG *Campus* Formiga

Formiga - MG
2019

Este trabalho é dedicado ao meu pai, minha mãe,
e a todos que de alguma forma me apoiaram para esta conquista.

AGRADECIMENTOS

Agradeço imensamente a Deus por me dar forças para a realização e a conclusão deste projeto.

À minha família, minha mãe que sempre me deu suporte e, ao meu falecido pai, que tragicamente morreu antes de me ver graduado, que sempre foi seu sonho, dedico este trabalho à eles.

À toda a equipe do IFMG, que conta com excelentes professores e funcionários que compartilham todo o seu conhecimento profissional e pessoal, em prol do crescimento dos discentes.

Aos meus professores, que hoje considero como amigos: Bruno Ferreira, Diego Mello, Everthon Valadão e Walace de Almeida.

Novamente agradeço ao professor Everthon Valadão que me auxiliou como orientador deste trabalho de conclusão de curso e ao professor Diego Mello, que sempre tratou os alunos com empatia, carinho e respeito, auxiliando quem está passando por dificuldades, nem sempre relacionadas à academia, eu sempre me espelharei em vocês.

Aos amigos que criei durante essa longa jornada, que são muitos, e que me auxiliaram desejando sempre o meu bem, pois sei que se trilhasse este caminho sozinho não teria ido tão longe como agora.

"The only thing ever achieved in life without effort is failure.
(Francis of Assisi)"

RESUMO

O presente trabalho de conclusão de curso apresenta o desenvolvimento de um *File System* (FS) focado em dispositivos de armazenamento que tem em seu domínio o acesso aleatório aos bytes. O projeto do FS baseou-se na criação dos objetos primários do Linux *Virtual File System*: Metadados do sistema de arquivos, Metadados do arquivo, Estrutura de diretórios, Alocação de blocos e Gerenciamento do espaço livre. Visando validar a persistência dos dados e aferir a vazão obtida, foi conduzida uma bateria de testes de escrita e de leitura, tanto sequencial quanto aleatória, utilizando tamanhos de arquivo de 10, 100 e 1000 MiBytes (representando diferentes cenários de uso). Tal avaliação de desempenho foi aplicada em três tipos de mídias de persistência de dados: SSD (*Solid State Drive*), *FlashDrive* – ambos de acesso aleatório – e em HDD (*Hard Disk Drive*), de acesso sequencial. Os resultados de desempenho do sistema de arquivos desenvolvido neste trabalho são considerados como preliminares, pois tal FS é experimentalmente utilizável em espaço de usuário (*user space*) uma vez que não foi efetivamente integrado ao *kernel* Linux (*kernel space*). Entretanto, a análise dos resultados sugere que o FS experimental auferiu benefícios do acesso aleatório em dispositivo SSD, tal qual o EXT4, porém não apresentou vazão compatível em dispositivo *FlashDrive* ("*PenDrive*"). Adicionalmente, este trabalho apresenta uma discussão e compreensiva análise de desempenho dos sistemas de arquivos NTFS (*New Technology File System*), EXT4 (*Fourth extended filesystem*) e BTRFS (*B-tree file system*), tendo sido neles conduzida a mesma metodologia de avaliação de desempenho.

Palavras-chave: Sistema de arquivos, Análise de desempenho, Dispositivos de armazenamento, Acesso aleatório

ABSTRACT

This present work shows the development of a File System (FS) focused on storage devices that has random access to bytes in its domain. The FS project was based on the creation of the Linux Virtual File System objects: File System Metadata, File Metadata, Directory Structure, Block Allocation, and Free Space Management. In order to validate the data persistence and verify the flow obtained, a battery of read and write tests, both sequential and random, was conducted using file sizes of 10, 100 and 1000 MiBytes (representing different usage scenarios). This performance assessment was applied to three types of data persistence media: Solid State Drive (SSD), FlashDrive (random access), and HDD (hard disk drive), sequential access. The results of the file system performance developed in this work are considered preliminary, since such FS is experimentally usable in user space, since it was not effectively integrated with the kernel. However, the analysis of the results suggests that the experimental FS had random access benefits on an SSD device, such as EXT4, but did not present a compatible flow in the FlashDrive device (“ PenDrive ”). In addition, this paper presents a comprehensive discussion and performance analysis of the New File System (NTFS), EXT4 (fourth extended file system) and BTRFS (B-tree file system), and the same methodology for evaluating performance was conducted in them.

Keywords: File System, Performance Evaluation, Storage Devices, Random Access

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

SSD	*Solid State Drive*
SO	Sistema Operacional
FS	*File System*
VFS	*Virtual File System*
HDD	*Hard Disk Drive*
FUSE	*File System in User space*
MiB	*MebiByte*
KiB	*Kibibyte*
GiB	*Gibibyte*
MB	*Megabyte*
KB	*Kilobyte*
GB	*Gigabyte*
NTFS	*New Technology File System*
EXT4	*Fourth extended filesystem*
COW	*Copy on write*
SEC	Segundos
TADs	Tipo Abstrato de dado
BTRFS	*B-tree file system*
RND	*Random*

BIOS *Basic Input/Output System*

SB Superbloco

RAID *Redundant Array of Inexpensive Drives*

SUMÁRIO

1 INTRODUÇÃO

Qualquer operação realizada por um usuário via programa de aplicação necessita que todos os dados sejam salvos de forma consistente a fim de recuperar e compartilhar tais dados com outras aplicações. A configuração de como esses dados são salvos depende especialmente do sistema operacional, que neste contexto organiza e armazena essas informações nos dispositivos de memória não volátil, chamados de armazenamento secundários.

Segundo Machado F.; Maia (2007), os arquivos são gerenciados pelo sistema operacional de maneira a facilitar o acesso dos usuários ao seu conteúdo. A parte do sistema responsável por essa gerência é denominada sistema de arquivos.

Basicamente um sistema de arquivo é uma parte que contempla o SO (Sistema Operacional), permitindo que o usuário possa manipular, recuperar, alterar e salvar arquivos, de forma bem transparente e estável, pois no caso contrário de um sistema de arquivo estável este poderá acarretar em um corrompimento dos dados, de tal modo que possa tornar os arquivos inutilizáveis para determinados usuários e programas de aplicação.

> É através do sistema de arquivos que os usuários terão uma interface para armazenar e recuperar seus dados, de forma transparente quanto aos detalhes de implementação e organização. E [sic] é através dele também que os diferentes processos do sistema poderão executar tarefas sobre os arquivos ou compartilhá-los com outros processos [...]. (POSSAMAI, 1999).

Como o sistema de arquivo é uma parte integrada ao Sistema Operacional, para a criação de um novo FS (*File System*) seria necessário a integração e recompilação intermitente para possíveis testes de aptidão, a fim de torná-lo consistente para quaisquer operações que um FS demanda.

Love (2004) comenta em seu livro que o *Kernel* do Linux possui um subsistema chamado VFS (*Virtual File System*), que possibilita uma interface transparente ao usuário, independente do sistema de arquivos utilizado.

Considerando a popularização de dispositivos de armazenamento secundário, tais como *Solid State Drive* - SSD, *Pen Drives* e *MicroSD*, há uma oportunidade para buscar melhorias no *software* que gerencia o armazenamento dos dados, uma vez que estes dispositivos são de acesso aleatório e a maioria dos sistemas de arquivos nativos presentes no SO sempre consideram a gravação de dados de forma sequencial. Isso faz com que os sistemas de arquivos atualmente em uso consigam utilizar estes dispositivos, mas as

estruturas de dados e mecanismos de gerenciamento podem não ser a melhor forma de aproveitar o potencial de um dispositivo que utiliza esta tecnologia, em especial os SSDs.

> O SSD tem uma vida útil predeterminada, ao contrário do que é verificado nos HDs. Existe um limite sobre quantas operações de gravação e leitura podem ser feitas antes de o componente estragar e precisar ser trocado.(RENATO, 2017).

O presente trabalho de conclusão de curso tem como finalidade a proposta de criação de um novo sistema de arquivo para dispositivos de acesso aleatório na distribuição Linux, bem como uma análise comportamental e comparativa entre os principais FS disponíveis e mais usados na literatura.

1.1 Justificativa

Os SSDs (*Solid State Drives*) sempre foram de extremo fascínio pelo autor deste trabalho, uma vez que ele teve a oportunidade de presenciar sua eficiência em relação à rapidez de acesso a um arquivo, se comparado com a tecnologia presente nos HDDs (*Hard State Drives*). Como *Flash drives*, SD *cards* e SSDs são dispositivos de acesso aleatório e considerando que os Sistemas de Arquivos (*File Systems* ou FS) mais populares (ex.: NTFS, EXT3 e FAT32) foram inicialmente projetados para dispositivos com acesso sequencial (*e.g.*: HDD), há espaço para experimentação com estruturas de gerenciamento do FS que se beneficiem de um acesso aleatório.

Um outro fator, que motivou a exploração de um sistema de arquivo alternativo para o dispositivo em questão, foi devido ao fato da tecnologia ainda ser muito cara, havendo espaço para otimização na manipulação dos dados, uma vez que os FS existentes não abordam completamente técnicas para maximizar a performance em geral, tratando-o igualmente como o seu antecessor, os HDDs. Portanto, dada a relevância dos *Flash drives* no cenário tecnológico atual, esta tecnologia foi escolhida como objeto de estudo deste projeto.

> *If you want to build a new file system, don't plan for today's hardware. Plan for the hardware that will be available 10 years from now.*[1](MASON, 2018)

É dado também como uma grande motivação o estudo dos principais FS disponíveis atualmente no âmbito geral (Literatura e uso de mercado), para que possamos entendê-los e tomarmos como premissa uma abordagem para alcançarmos o êxito em nosso projeto. Para que isso se torne factível, serão necessários testes de desempenhos dos FS com o

[1] Se você deseja criar um novo sistema de arquivos, não planeje para o *hardware* de hoje. Planeje para o *hardware* que estará disponível daqui a 10 anos. Tradução nossa.

intuito de compará-los para denotar qual se sobressai, a fim de apurar e visualizar seu comportamento em diferentes tecnologias utilizadas *(eg.: Flash Drive, HDD e SSD).*

Por fim, ao projetar e implementar um sistema de arquivos experimental, o trabalho proposto permitirá aplicar diversos conceitos aprendidos no curso como, por exemplo: projeto e análise de algoritmos, estrutura de dados, sistemas operacionais, arquitetura e organização de computadores, dentre vários outros recursos aprendido ao longo do curso.

1.2 Objetivos

1.2.1 Objetivo Geral

Projetar e implementar um Sistema de Arquivos (FS) experimental em *userspace*, para gerenciar um volume para fins de armazenamento persistente, com foco na sua utilização em dispositivos de armazenamento do tipo SSD (*Solid State Drives*). Este trabalho explora abordagens para as estruturas básicas de um *Linux Virtual File System*, tais como estruturas do volume e estruturas de diretórios. Ainda, serão abordados mecanismos de alocação de blocos e gerenciamento do espaço. O principal ponto a ser abordado neste TCC é experimentar técnicas alternativas para gerenciamento de FS que explorem o acesso aleatório possibilitado por SSDs, com foco em melhorias no desempenho da manipulação de arquivos e buscando preservar a vida útil do dispositivo.

1.2.2 Objetivos Específicos

O Sistema de Arquivos (FS) a ser desenvolvido terá como foco dispositivos de armazenamento de acesso aleatório *(e.g., Flash Drive, Solid State Drive)*, tendo como objetivos específicos:

1. Implementar a interface VFS (*Linux Virtual File System*), para sua utilização em um sistema operacional Linux, contendo estruturas do volume (*superblock, inode, file*) e estrutura de diretórios (*dentry*);

2. Projetar e implementar mecanismos de FS apropriados para armazenamento com acesso aleatório, no que tange ao mecanismo de alocação de blocos e gerenciamento de espaço livre;

1.3 Estrutura do Trabalho

Os tópicos abordados que serão apresentados a seguir neste trabalho se dividem da seguinte forma:

- Na Introdução, mostramos brevemente o que é o trabalho e o que de fato ele desenvolve, os objetivos, a motivação, bem como sua estrutura.
- O Segundo capítulo diz respeito à fundamentação teórica, onde será descrito toda base necessária para o início e desenvolvimento das atividades deste projeto, descrevendo todos os conceitos estudados, tais eles como: Sistemas de Arquivos, VFS (*Virtual File System*), FUSE (*File System in User space*), EXT4 (*Fourth extended filesystem*), BTRFS (*B-tree file system*) e NTFS (*New Technology File System*) e alguns trabalhos relacionados que ajudaram para a criação e realização deste projeto.
- O Terceiro capítulo tem como finalidade mostrar os materiais e os métodos usados neste trabalho.
- Posteriormente o quarto capítulo demonstra a parte do desenvolvimento do projeto, detalhando o mesmo exibindo suas estruturas.
- A quinta parte tem como objetivo exibir os resultados coletados bem como suas análises.
- Por fim, o capítulo seis exibe uma conclusão do trabalho desenvolvido, e o que poderá ser feito posteriormente para complementar o desenvolvimento deste projeto (trabalhos futuros).

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo contempla com todos os conceitos estudados para a realização do trabalho proposto, tais eles como: Sistemas de Arquivos, VFS (*Virtual File System*), FUSE (*File System in User Space*), FS escolhidos (EXT4, BTRFS e NTFS), e também alguns trabalhos relacionados que ajudaram para a criação e realização deste trabalho.

2.1 Sistema de Arquivos

Silberschatz (2009) afirma que, para a maioria dos usuários, o sistema de arquivos é a parte mais visível de um sistema operacional pois ele fornece o mecanismo para o armazenamento tanto dos dados do utilizador quanto dos programas. Lima (2012) define que um sistema de arquivo é a parte do SO responsável pelo gerenciamento de arquivos (estrutura, identificação, acesso, utilização, proteção e implementação). Já Cruz (2015) diz que os sistemas de arquivos fornecem um instrumento que permite os usuários armazenarem seus dados em arquivos e diretórios, organizados em uma estrutura hierárquica. Sintetizando, um Sistema de Arquivos (ou *File System* – FS) corresponde a uma coleção organizada (geralmente exibida em formato de árvore) de arquivos e diretórios.

O sistema em questão tem como finalidade atuar com uma interface amigável e simples entre os dados (arquivos) e o usuário, através da qual o utilizador pode interagir com esses arquivos, manipulando-os conforme desejar, seja alterando um nome do arquivo ou não permitindo que usuários do mesmo computador tenha acesso a dados de uma outra conta.

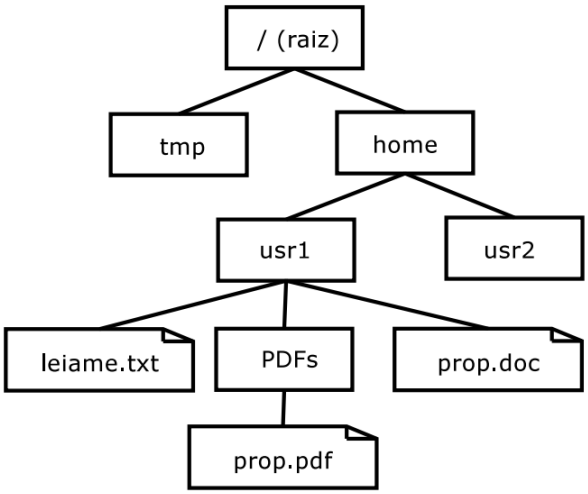
> O sistema de arquivos é uma parte importantíssima do sistema operacional, pois ele fornece uma visão abstrata dos dados persistentes (também chamado de armazenamento secundário), além de ser responsável pelo serviço de nomes, acesso a arquivos e de sua organização geral. (CARVALHO, 2005)

Os sistemas de arquivos consistem em duas partes distintas: uma coleção de arquivos, cada um deles armazenando dados relacionados, e uma estrutura de diretórios, que organiza e fornece informações sobre todos os arquivos do sistema (Silberschatz, p. 221).

Silberschatz (2009) conta que cada dispositivo em um sistema de arquivos contém um índice de volume ou um diretório que lista a localização dos arquivos no dispositivo, a fim de criar diretórios para permitir a organização dos dados, tanto para o sistema, tanto para a comunicação com o usuário.

A Figura 1 exemplifica como é uma árvore de diretórios nas distribuições Linux, a fim de simplificar a compreensão da estrutura hierárquica de um sistema de arquivos que nele é contemplado, contendo pastas, documentos e o diretório da raiz que é onde toda estrutura começa.

Figura 1 – Dentry (Árvore de diretórios de um FS)



Fonte: (CARVALHO, 2005)

Em toda hierarquia, no nó folha dessa estrutura de dados, temos um arquivo atrelado à um diretório, que são dados propriamente salvos pelo o usuário ou por aplicações que podem ser constantemente alterados.

> Tiago Flach (2009, p. 16) classifica como uma das principais operações de um sistema de arquivos, a operação de montagem. Segundo ele a montagem é uma associação entre um desses sistemas e um dispositivo de armazenamento secundário. O comando *mount* é utilizado para anexar um sistema de arquivos à hierarquia do sistema atual (raiz). Durante uma montagem comum no Linux, é necessário fornecer um tipo de sistema de arquivos, um sistema em si e um ponto de montagem.(FLACH, 2009)

Tanenbaum Andrew S.; Woodhull (2008) dizem que os arquivos são um mecanismo de abstração. Eles proporcionam uma maneira de armazenar informações no disco e de lê-las posteriormente. Isso deve ser feito de modo a esconder do usuário os detalhes sobre como e onde as informações são armazenadas e como os discos realmente funcionam.

Já Silberschatz (2009) conta que um arquivo é um tipo de dado abstrato definido e implementado pelo sistema operacional. É uma coleção nomeada de informações relacionadas registradas em memória secundária. Da perspectiva do usuário, um arquivo é a menor unidade de armazenamento.

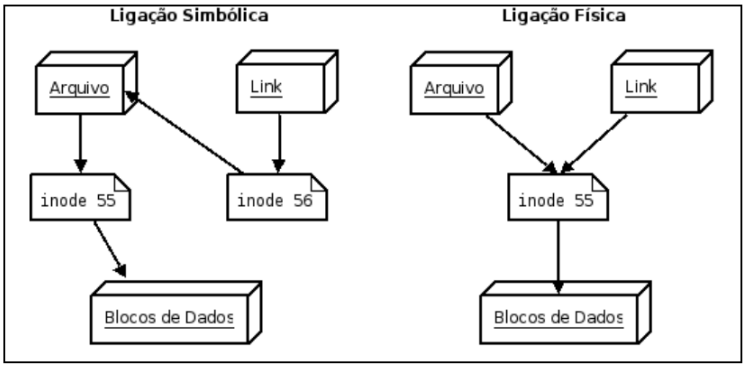
Segundo Love (2004) os sistemas *Unix* separam o conceito de arquivo de qualquer informação relacionada a ele (permissões de acesso, tamanho, proprietário, entre outros atributos). Essas informações normalmente são chamadas de metadados e são armazenados em uma estrutura de dados separada do arquivo, chamada de *inode* (*index node*).

Todo arquivo contém sua peculiaridade que são os metadados. Em um computador podem existir diversos arquivos com diversos conteúdos caracterizados pelo nome que é um atributo de chave primária, ou seja, dois ou mais arquivos não podem habitar com o mesmo nome no mesmo diretório na maioria dos SO existentes. Os atributos de um arquivo (metadados que contemplam a estrutura de um *inode*) variam de um sistema operacional para outro, mas normalmente segundo Silberschatz (2009), os mais comuns são :

- **Nome**. O nome simbólico do arquivo é a única informação mantida em forma legível pelas pessoas;
- **Identificador**. Este rótulo exclusivo, usualmente um número, identifica o arquivo dentro do sistema de arquivos; é um nome do arquivo não legível pelas pessoas.
- **Tipo**. Esta informação é um ponteiro para sistemas que suportam diferentes tipos de arquivos.
- **Localização**. Esta informação é um ponteiro para um dispositivo e para a localização do arquivo neste dispositivo.
- **Tamanho**. O tamanho corrente do arquivo (em *bytes*, palavras ou blocos) e possivelmente o tamanho máximo permitido são incluídos neste atributo.
- **Proteção**. A informação de controle de acesso determina quem pode ler o arquivo, gravá-lo, executá-lo e assim por diante.
- **Hora, data e identificação do usuário**. Estas informações podem ser mantidas para a criação, última modificação e última utilização do arquivo. Estes dados podem ser úteis para proteção, segurança e monitoramento do uso do arquivo.

Conforme conta Flach (2009) como o SO faz um acesso a um *inode*, ele conta que em um primeiro momento o sistema operacional procura na tabela de *inodes*, o *inode* do arquivo, de posse deste obtém os metadados do arquivo em questão e assim consegue encontrar a localização dos dados no disco físico. A Figura 2 ilustra como são feitos esses acessos.

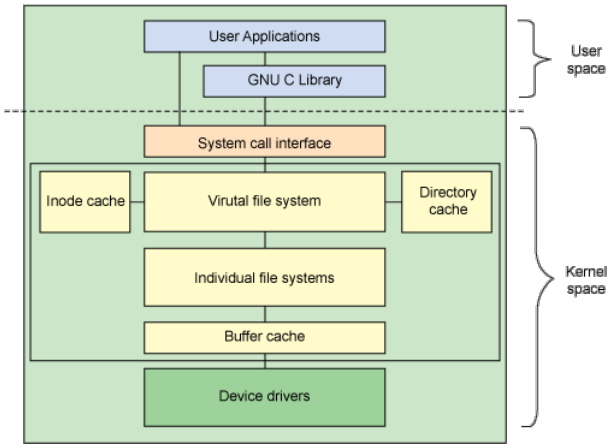
Figura 2 – Exemplo do acesso dos dados em uma memória secundária



Fonte: (FLACH, 2009)

Jones (2007), diz que os componentes do sistema de arquivos de um SO dividem entre operações entre nível usuário e operações feitas pelo *Kernel*, uma ilustração de como isso é feito é mostrado pela Figura. 3.

Figura 3 – Divisões de operações de um Sistema de arquivo Linux.



Fonte: (JONES, 2007)

Ainda assim, Jones (2007) comenta sobre as estruturas do sistema de arquivos Linux que são definidas da seguinte forma:

1. **VFS:** atua como o nível raiz da interface do sistema, mantendo o rastreio dos sistemas de arquivos suportados atualmente, bem como dos que estão sendo montados no momento;

2. ***Superblock*:** estrutura que representa um sistema de arquivos. Essa estrutura armazena as informações necessárias para gerenciar o sistema durante a operação. Inclui o nome do sistema de arquivos (ex.: *ProjetoFileSystem*), o tamanho do sistema de arquivos e seu estado, uma referência para o dispositivo de bloco e informações

de metadados (como listas livres, etc). Essa estrutura geralmente é armazenada na mídia de armazenamento, mas pode ser criada em tempo real se não existir nenhuma;

3. ***Inode*:** representa um objeto no sistema de arquivos com identificador exclusivo. Essa estrutura contém todos os metadados do objeto em questão, como seu dono, suas permissões, suas datas, entre outros;

4. ***Dentry*:** estruturas utilizadas na conversão entre nomes de arquivos e *inodes*;

5. **Cache do *Buffer*:** mantém o rastreio de pedidos de leitura e gravação de implementações do sistema de arquivos individual e dispositivos físicos (através de *drivers* de dispositivo). Para eficiência, o Linux mantém um cache dos *buffers* utilizados recentemente para evitar ter que voltar para o dispositivo físico em todos os pedidos.

Para a criação de novos Sistemas de arquivos uma nova camada foi criada nos periféricos Linux, uma interface genérica entre o *Kernel* e os sistemas de arquivos, chamada VFS. Com ela programadores podem desenvolver um FS do seu modo e integrá-lo no Sistema Operacional de uma forma mais abstrata, com a condição que o desenvolvedor respeite as assinaturas das funções imposta pelo VFS.

2.2 VFS (*Virtual File System*)

Como citado anteriormente o *Virtual File System* é uma camada de *software* embutido no *Kernel Linux* que trata todas as chamadas de sistemas referentes a um sistema de arquivos tais elas como *open*, *read* e *write*. Sua principal característica é prover uma interface genérica englobando diversos tipos de sistemas de arquivos que podem usar sua estrutura independentemente do sistema de arquivos usados ou do meio físico.

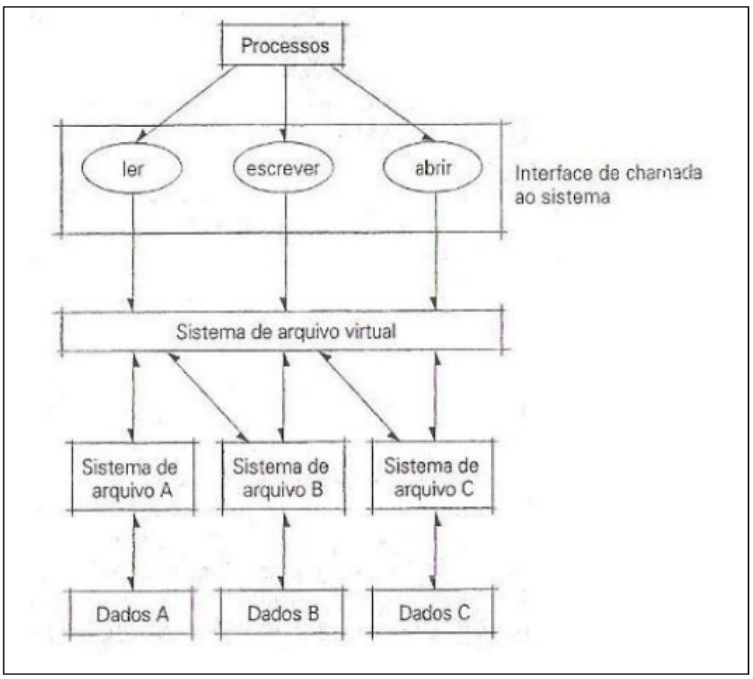
Andrey (2005) conta que os sistemas de arquivos são implementados de forma a prover as abstrações, tanto de interface quanto de estrutura de dados, que a camada VFS espera. Um exemplo desta operação é quando é usado a função *write*() à nível de usuário ilustrada na Figura 4.

Figura 4 – Uso de funções de nível usuário para o VFS.

Fonte: (ANDREY, 2005)

Segundo Flach (2009) é o VFS quem permite aos sistemas de arquivos, efetuarem chamadas como *open*, *read* e *write* ao manipular um arquivo, sem que tenha que implementar em sí as operações de baixo nível. Efetuando essas chamadas um desses sistemas pode, através do VFS, se comunicar com diferentes sistemas de arquivos e meios, efetuando operações entre sí, a Figura 5 ilustra este processo em questão.

Figura 5 – Exemplo de procedimentos em sistema Linux.



Fonte: (FLACH, 2009)

Flach (2009) também cita algumas das principais funções disponibilizada no VFS para trabalhar com diretórios e arquivos manipulando-os conforme desejado. Algumas destas funções é apresentada na Tabela 1.

Tabela 1 – Principais funções do VFS

Função	**Descrição da Função**
lseek	Atualiza o ponteiro do arquivo para o deslocamento do dado.
read	Lê os dados de um arquivo.
write	Escreve os dados em um arquivo.
readdir	Retorna o próximo diretório em uma listagem de diretórios.
open	Cria um novo objeto de arquivo e entra ao objeto inode correspondente.
release	Esta função é chamada pelo VFS quando a última referência restante para o arquivo é destruída.
lock	Manipula um bloqueio do arquivo.
sendfile	Copia dados de um arquivo para outro.
sendpage	Envia dados de um arquivo para outro.

Fonte: (FLACH, 2009)

Tendo em vista tudo que foi abordado, o VFS provê a estes sistemas de arquivos um *framework* para a sua implementação e uma interface para que trabalhem com as chamadas de sistema padrão do Linux.

2.3 FUSE (File System in User Space)

O FUSE é um módulo que pode ser instalado no Kernel do Linux, para possibilitar o desenvolvimento de sistemas de arquivos, que executam no espaço do usuário. As principais vantagens do uso do FUSE são (FUSE, 2009):

a) possuir uma API simples;

b) possuir um processo simples de instalação (não necessita da aplicação de atualizações ou da re-compilação do Kernel);

c) possuir uma implementação segura;

d) permitir a execução dos sistemas no espaço do usuário, pois disponibiliza uma interface muito eficiente com o Kernel;

e) ser utilizável por usuários sem privilégios;

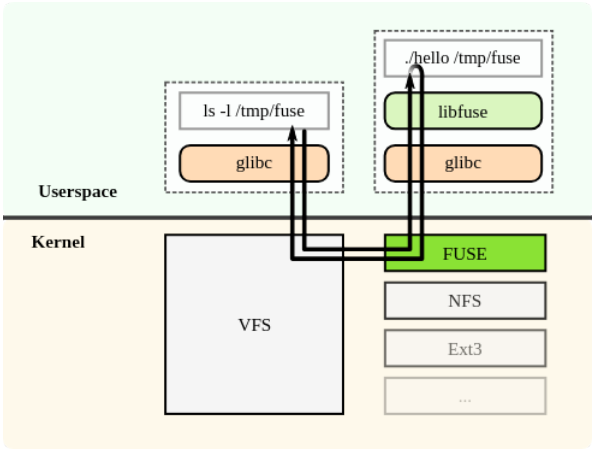
f) ser compatível com as versões do Kernel 2.4 e 2.6;

g) ser muito estável.

Basicamente o FUSE é composto por três componentes principais que são a base para o seu funcionamento. O primeiro item é um módulo do Kernel do sistema que se

apresenta perante o VSF como um novo sistema de arquivos, posteriormente esse módulo disponibiliza um *device* para a comunicação dos processos do nível do utilizador. Como segundo componente temos a biblioteca *libfuse* que se comunica com o *device* criado. Por fim temos a definição do sistema de arquivos, que com recurso da biblioteca consegue registrar um ponto de montagem e define um conjunto de *handlers*. (LOPES, 2009). A Figura 6 tem como finalidade ilustrar como esse processo é feito.

Figura 6 – Espaço FUSE.



Fonte: (LOPES, 2009)

Lopes (2009) também conta que no nível utilizador, o FUSE disponibiliza uma biblioteca que comunica com o núcleo do kernel, aceitando os pedidos vindos do *device* e fazendo a sua tradução para chamadas de funções similares às da interface do VFS. Essas funções possuem nomes como *open*() que abre um arquivo, *read*() que lê um tamanho em bytes de um arquivo, *write*() que escreve um tamanho em bytes de um arquivo, *rename*() que renomeia um diretório, arquivo entre outro objeto. Tais outras operações podem ser vistos na Figura 7 que demonstra outras funções do FUSE.

Figura 7 – Operações do FUSE.

```
struct fuse_operations {
    int (*getattr) (const char *, struct stat *);
    int (*readlink) (const char *, char *, size_t);
    int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
    int (*mknod) (const char *, mode_t, dev_t);
    int (*mkdir) (const char *, mode_t);
    int (*unlink) (const char *);
    int (*rmdir) (const char *);
    int (*symlink) (const char *, const char *);
    int (*rename) (const char *, const char *);
    int (*link) (const char *, const char *);
    int (*chmod) (const char *, mode_t);
    int (*chown) (const char *, uid_t, gid_t);
    int (*truncate) (const char *, off_t);
    int (*utime) (const char *, struct utimbuf *);
    int (*open) (const char *, struct fuse_file_info *);
    int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
    int (*write) (const char *, const char *, size_t, off_t,struct fuse_file_info *);
    int (*statfs) (const char *, struct statfs *);
    int (*flush) (const char *, struct fuse_file_info *);
    int (*release) (const char *, struct fuse_file_info *);
    int (*fsync) (const char *, int, struct fuse_file_info *);
    int (*setxattr) (const char *, const char *, const char *, size_t, int);
    int (*getxattr) (const char *, const char *, char *, size_t);
    int (*listxattr) (const char *, char *, size_t);
    int (*removexattr) (const char *, const char *);
};
```

Fonte: Elaborado pelo autor.

Após compactuar com o padrão de como as funções do FUSE trabalha, a biblioteca dá ao usuário a disponibilidade de chamá-las do jeito que bem entender, mapeando as funções implementadas no que elas realmente irão executar, desde de que o usuário respeite as assinaturas apresentadas na Figura 7. A Figura 8 ilustra um exemplo da estrutura em questão, onde para cada primitiva usada pelo sistema de arquivos é vinculada com uma função escrita pelo usuário.

Figura 8 – Estrutura de registro de funções definidas no espaço do usuário.

```
struct fuse_operations FileSystem_oper = {
        .mknod = FileSystem_mknod,
        .mkdir = FileSystem_mkdir,
        .rename = FileSystem_rename,
        .open = FileSystem_open,
        .read = FileSystem_read,
        .write = FileSystem_write,
}
```

Fonte: Elaborado pelo autor.

Com o objetivo principal de promover um sistema de arquivos a nível de usuário, o FUSE é um *middleware* que serve de interface entre o Kernel e o ambiente do usuário, fazendo com que todas as chamadas do sistema compactuem com a interface do VSF.

2.4 EXT4 (*Fourth Extended Filesystem*)

Sucessor do EXT3, o sistema de arquivos EXT4, criado por Theodore Ts'o, conta com uma série de melhorias importantes mediante ao seu antecessor em sua estrutura destinada ao armazenamento de dados. Isso resultou em um FS com um design aperfeiçoado, melhoria de performance, confiável e com muitos outros recursos.

Dentro das suas principais funcionalidades, o EXT4 conta com alocação tardia, Jornal *checksumming*, suporte para tamanhos maiores de volumes e arquivos e compatibilidade com versões anteriores, Cruz (2015) descreve suas funcionalidades como segue:

- **Compatibilidade entre Versões Futuras e Antigas:** Essa funcionalidade possibilita a conversão do sistema de arquivo EXT3 para o EXT4 sem que haja a necessidade de formatação da partição, e vice versa.
- **Melhoramentos no Intervalo de Tempo e no Registro de Data e Hora:** A hora e a data presente no EXT4 está mais precisa, pois conta com a evolução de segundos para nanosegundo. O sistema de arquivo também conta com dois bits a mais que seu antecessor o que possibilita o aumento da data máxima que pode ser atribuída em um arquivo.
- **Escalabilidade:** Usando blocos de 4KB, o EXT4 suporta arquivos únicos de até 16 TB (terabyte) e gerencia partições de até 1 EB (exabyte). E ainda, o limite de subdiretórios foi expandido de 32.000 (trinta e dois mil) para virtualmente ilimitados;
- **Implementação de Extensões:** O uso de extensões aumenta o desempenho e facilita o controle do sistema de arquivo. As extensões são áreas que marcam os limites de blocos contínuos, o que melhora muito o desempenho no controle de arquivos grandes que ocupam vários blocos;
- **Desempenho:** O EXT4 apresenta inúmeras melhorias de desempenho, dentre as quais se destacam:
  - Pré alocação de Nível de Arquivo: O EXT4 reserva uma quantidade de blocos sequenciais, pré alocando estes blocos por meio de uma chamada do sistema. Após esta operação ele grava os dados desejados nestes blocos adjacentes, otimizando a leitura desses dados;
  - Alocação dos Blocos com Atraso: O EXT4 automaticamente atrasa a alocação dos blocos físicos no disco. Desse modo, mais dados são apresentados para alocar e gravar em blocos adjacentes.
  - Alocação de Vários Blocos: O EXT4 consegue alocar vários blocos em uma única chamada. Assim, aumenta a probabilidade de utilizar blocos adjacentes no disco, reduzindo o processamento requerido para a alocação dos blocos.

- **Confiabilidade:** Para melhorar a confiabilidade, o EXT4 possui mecanismos de autoproteção e autorreparo, dos quais se destacam:
  - *Journaling* do Sistema de Arquivos: O EXT4 é um sistema de arquivos jornalado usando como base arquivos log, assim sendo, ele possui um processo de registro das mudanças que ocorrem no sistema de arquivos. O journal do EXT4 pode operar em três modos: *Writeback*; *Ordered*; e *Jornal*;
  - Verificação do *Journal* com *Checksum*: O EXT4 realiza cálculos para checar os dados do *journal* mantendo assim o FS íntegro;
  - Desfragmentação *Online*: Para reduzir a fragmentação existe o *e4defrag*, que é uma ferramenta de desfragmentação *online*.

Já consolidado, o EXT4 é o principal sistema de arquivos usado nas distribuições Linux, devido sua alta gama de funcionalidades bem como sua confiabilidade e estabilidade.

2.5 BTRFS (*B-Tree File System*)

Inicialmente desenvolvido pela Oracle Corporation, o BTRFS é um sistema de arquivos que tem como princípio a cópia em gravação (COW), que basicamente é uma técnica de gerenciamento de recursos usada na programação de computadores para implementar eficientemente uma operação "duplicar"ou "copiar"em recursos modificáveis (KASAMPALIS, 2010). Usando a técnica COW, o BTRFS tem uma abordagem bastante eficiente em dispositivos de acesso aleatório, evitando possíveis escritas desnecessárias, podendo assim melhorar a vida útil do periférico utilizado.

O BTRFS, segundo Praciano (2016), conta com uma série de funcionalidades, dentre as principais estão elas:

- **Metodologia de escrita e acesso:** Utilizando uma estrutura do tipo B-Tree, o BTRFS armazena tipos de dados e aponta para informações dentro de uma unidade. Diferente de outros sistemas de arquivos, não faz *journaling* de transações. O resultado disso é uma escrita feita uma única vez, o que remove limitações consequentes do espaço ocupado pelo *journal* e reduz desgaste causado por gravações repetitivas na mesma seção do disco rígido ou do SSD. A técnica *copy-on-write* garante que blocos e extensões não sejam sobrescritos;
- **Ferramenta de conversão de sistemas de arquivos para o EXT:** No BTRFS existe a possibilidade da conversão local de sistemas de arquivos EXT3 e EXT4. Os metadados originais dos sistemas antigos são mantidos em um *snapshot* que basicamente é uma “foto” tirada de um diretório, de forma que a conversão possa ser revertida, caso seja necessário;

- **Uso de Subvolumes:** Os subvolumes no BTRFS são sub-árvores do sistema de arquivos primário que basicamente são árvores criadas sobre o sistema raíz primário e que podem ser tratadas como um sistema de arquivos separado. O seu próprio ponto de montagem contém opções e políticas. Diferente da criação de múltiplas partições de disco, subvolumes não requerem alocação adicional no disco. Eles são apenas diretórios vazios até que você comece adicionar arquivos a eles.

2.6 NTFS (*New Technology File System*)

Criado pela Microsoft em 1993, o NTFS é um sistema de arquivos construído à base de arquivos. Segundo Hudson (2010), a principal abordagem usada pelo NTFS é a tabela mestra de arquivos, ou MFT (*Master File Table*), sendo este a base de todo sistema, pois abrange todos os metadados que por sua vez contém ou aponta para todos os outros arquivos contidos no FS. Hudson (2010) também conta que a única exceção é o registro mestre de inicialização (MBR - *Master boot record*), que é uma partição do sistema, e não um arquivo como os outros demais e precisa ser carregado pelo SO para que o sistema de arquivo seja inicializado. A Figura 9 tem como finalidade exemplificar uma abstração do MFT do FS em questão.

Figura 9 – Master File Table.

Metadata File Name	Filename	Description
Master File Table	$MFT	The Master File Table, holder of NTFS metadata
Master File Table Copy	$MFTMirr	A backup copy of the first 16 records of the MFT
Log File	$LogFile	The journaling log
Volume Descriptor	$Volume	Contains volume/partition information like name, version
Attribute Definition Table	$AttrDef	A table of attribute definitions used in a volume
Root Directory	. - a dot	A pointer to the root directory of the volume
Cluster Allocation Bitmap	$Bitmap	A binary map of which clusters are used and which are free
Volume Boot Code	$Boot	A copy or a pointer to the volume boot code.
Bad Cluster File	$BadClus	A list of disk clusters that are marked as bad, not to be used
Quota Table	$Quota	A table containing disk quota info.
Upper Case Table	$UpCase	A table containing info for converting file names to Unicode

Fonte: (HUDSON, 2010)

Dentro dos FS existentes na época de criação, o NTFS já se sobressaia devido suas novas funcionalidades. Após atualizações, o FS em questão ficou mais consolidado e menos suscetível a falhas. Segundo a Microsoft atualmente ele fornece um conjunto completo de recursos tais como descritores de segurança, criptografia, cotas de disco e metadados sofisticados. Em um contexto geral, as suas principais aplicações práticas deste FS são descritas a seguir (MICROSOFT, 2009):

- **Mais confiabilidade:** Para restaurar a integridade do sistema de arquivos quando o computador for reiniciado após uma falha do sistema, o NTFS usa suas informações de arquivo e o ponto de verificação a partir dos arquivos de *log*, a fim de recuperar o conteúdo corrompido;

- **Suporte para grandes volumes de dados:** O NTFS pode oferecer um suporte a volumes de até 256 terabytes. Os suportes em questão são afetados pelo tamanho do *cluster* e o número de *clusters* do volume.
- **Aumento da segurança de lista de controle de acesso para arquivos e pastas:** O NTFS permite que você defina permissões em um arquivo ou pasta, especificando os grupos e usuários cujo acesso você deseja restringir ou permitir.

O suporte ao NTFS tem aumentado muito devido ao fato de ser um FS que conta com diversas funcionalidades, pois por mais que sua patente seja de distribuições Windows, a comunidade do Linux conseguiu embuti-lo em seu Kernel, tornando-o assim um FS escalável.

2.7 Trabalhos Relacionados

Para a realização deste projeto foram necessário o levantamento de diversos artigos e pesquisas sobre FS existentes a fim de se extrair algum conteúdo que de alguma forma estivesse diretamente ou indiretamente relacionado ao nosso objetivo alvo. Dentre inúmeros artigos, dois deles foram de cunho importante para a nossa implementação, *Introduction to Flash Memory* (ROBERTO, 2003) e *NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories* (JIAN, 2016).

No artigo que se diz respeito a *Flash Memory*, pudemos observar como é feito o acesso aos seus dados, uma vez que todo SSD é constituído por inúmeros chips de *Flash Memory*. Ao observar como era o comportamento destes dados neste hardware, o autor deste projeto pode tomar uma iniciativa de como as informações irão ser salvas no FS em desenvolvimento, sendo tomada a atitude de separação de fatias, para o aproveitamento do hardware em questão. O artigo também forneceu à este projeto informações sobre tempo de vida de um dispositivo de *Flash Memory* alertando que devemos tomar cuidado na quantidade de operações de escrita que serão executadas neste dispositivo, pois quanto mais operações de escritas feitas, menor é o seu tempo de vida. (ROBERTO, 2003)

O artigo *NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories*, Jian (2016) conta sobre um sistema de arquivos para dispositivos que usam memória do tipo *Flash*. O principal tópico desse artigo que compactua com as ideias para a realização deste trabalho é como o autor trata a solução de vida útil do dispositivo utilizado. A solução dada é a criação de um sistema híbrido constituído de Memória *Flash* (SSD) e Memória com disco magnético (HDD), onde os arquivos de log são salvos no HDD e os arquivos com seus metadados são salvos no dispositivo principal SSD (JIAN, 2016).

2.8 Maturidade Tecnológica

Segundo Moresi (2017), no Plano Tecnológico da NASA[1] tecnologia é definida como "a aplicação prática do conhecimento da capacidade de fazer algo de forma inteiramente nova". A avaliação da maturidade de uma tecnologia é necessária pois evita o comprometimento das aplicações com seus orçamentos e tempo disponível, devendo ser feitas repetidas vezes até que os requisitos e recursos estejam dentro do risco aceitável. A avaliação das tecnologias pode assim funcionar como componente de avaliação técnica global e de gestão de riscos.

2.8.1 Prova de Conceito, Protótipo e MVP

Em uma etapa intermediária entre a formulação de uma solução e sua avaliação, pode-se construir modelos práticos preliminares para verificar os aspectos chave do projeto. Uma prova de conceito é um projeto de pequeno porte, construído para verificar se determinada característica pode ser realizada na prática, sem se preocupar com a usabilidade. Também conhecida como PoC (*Proof of Concept*) pode ser aplicada logo no início de um projeto (antes mesmo do protótipo) provando que o mesmo é viável ou não, evitando assim o desperdício de tempo e recursos. Ainda que simples, a prova de conceito pode ser um mecanismo estimulador e que mostre as reais possibilidades do projeto (MOREIRA, 2017).

Já um protótipo é um sistema computacional em desenvolvimento, mostrando como determinadas características serão realizadas. Um protótipo é um modelo funcional e interativo do produto final, utilizado como uma versão inicial de um sistema (isto é, não completo), que auxilia a conhecer melhor os desafios enfrentados em um ambiente realístico. O protótipo também pode ser utilizado para testar um produto ou serviço e também servir de base para mudanças e novas implementações (ENDEAVOR, 2015).

Por fim, um Produto Minimamente Viável (MVP) é a forma mínima de um produto testado em ambiente real com usuários finais. Enquanto um protótipo permite corrigir problemas identificados em um estágio inicial do produto, um MVP busca identificar possíveis melhorias no produto do ponto de vista do usuário final. O intuito de um MVP é colocar a prova o real intuito do projeto. Não é o produto final a ser entregue pelo projeto mas sim um que contenha o mínimo de recursos possíveis para funcionar e mostrar o seu valor (EMPREENDEDORA, 2017).

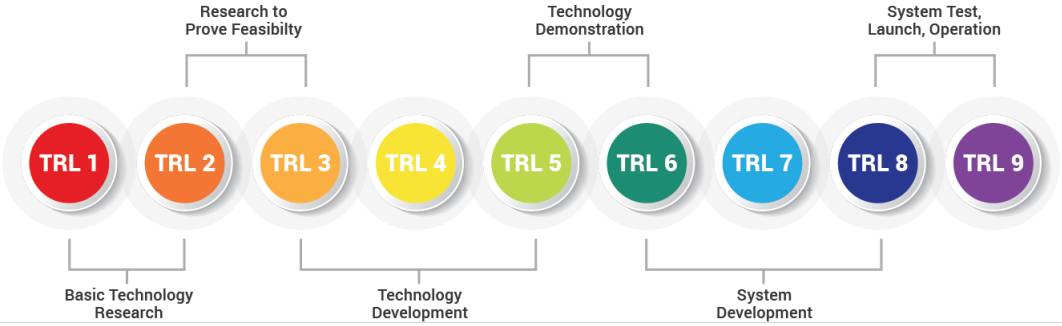
2.8.2 Nível de Maturidade Tecnológica (TRL)

O nível de maturidade tecnológica ou TRL (*Technology Readiness Level*) é uma medida concebida por Stan Sadin, pesquisador da NASA, e consiste em uma escala de

[1] <http://www.hq.nasa.gov/office/codeq/trl/trlchrt.pdf>. Acessado em abr/19

nove níveis que categorizam a maturidade tecnológica de um sistema, sendo o nível nove a tecnologia mais madura (EMBRAPII, 2016).

Figura 10 – Fases de maturidade de uma tecnologia na escala TRL.



FONTE: Abaco Systems

A Figura 10 sintetiza as principais fases do desenvolvimento de uma tecnologia e a Tabela 2 apresenta a definição de cada nível TRL[1].

Tabela 2 – Definição dos níveis de maturidade na escala TRL

PRODUÇÃO	TRL 9: Produto completo, pronto para implantação/comercialização em larga escala e disponível aos usuários finais.
	TRL 8: Pré-produção do sistema completo, com processos validados e ainda passível de ajustes.
VALIDAÇÃO	TRL 7: Demonstração de sistema piloto em ambiente necessário/operacional (teste de campo).
	TRL 6: Validação do protótipo de sistema com demonstração das funções críticas, em ambiente relevante (experimento em cenário virtual/sintético).
PROTÓTIPO	TRL 5: Teste laboratorial da integração dos componentes, em ambiente relevante (experimento em cenário virtual/sintético).
	TRL 4: Teste de bancada de componente ou processo (validação funcional) em ambiente laboratorial (pequena escala).
IDEIA	TRL 3: Prova de conceito analítica e experimental, das características ou funções críticas (teste de viabilidade)
	TRL 2: Formulação da tecnologia, conceito ou aplicação.
	TRL 1: Princípios básicos observados e documentados.

Considerando a escala TRL de maturidade tecnológica, esse projeto de conclusão de curso visa atingir o nível de maturidade TRL-5: teste laboratorial da integração dos componentes, em ambiente relevante (experimento em cenário virtual/sintético).

[1] <https://www.abaco.com/technology-readiness-level>. Acessado em abr/19

3 MATERIAIS E MÉTODOS

Nesta seção serão descritos todos materiais usados na realização deste TCC, bem como a metodologia usada.

3.1 Materiais

Nesta subseção serão descritos os *hardwares* e *softwares* que foram necessários para a execução do trabalho.

3.1.1 *Hardware*

A Tabela 3 tem como finalidade demonstrar os periféricos utilizados para a execução deste trabalho.

Tabela 3 – Materiais utilizados

Item	**Descrição**
Hardware	Notebook Acer F5-573G-75A3
	Processador Intel Core i7-7500U 2.7GHz com Turbo Boost para 3.5GHz
	Memória RAM 8GB DDR4
	HDD 1TB
	PenDrive Kingston 8GB
	SSD M2 WD Green 240GB
Sistema Operacional	Elementary OS 0.4.1 64 bits
	Linux Kernel 4.x
IDE	Sublime Text 2.0
	NetBeans IDE 8.2
Linguagem de programação	C / R / Bash

Fonte: Elaborado pelo autor.

3.1.2 *Software*

Aqui será brevemente descrito os *softwares* usados para a realização deste trabalho.

3.1.2.1 IOzone Filesystem Benchmark

Criado por William Norcott et al, o IOzone[1] é um *software open source* que têm como finalidade a avaliação de desempenho de um sistema de arquivo. Basicamente ele lê e grava dados em um dispositivo, obtendo sua vazão em KB/sec de escrita e leitura sequencial, bem como aleatória. Seus parâmetros são muitos, mas para este projeto foram utilizados os seguintes argumentos descritos na Tabela 4.

Tabela 4 – Parâmetros do *software* IOzone.

-i	Usado para especificar quais testes são executados (e.g. 0 = escrever/reescrever, 1 = ler/reler, 2 = ler/escrever aleatoriamente).
-I	Diz ao sistema de arquivos que todas as operações são para ignorar o cache de *buffer* e ir diretamente para o disco.
-s	Usado para especificar o tamanho em Kbytes, do arquivo a ser testado.
-r	Usado para especificar o tamanho do registro, em Kbytes, para o teste.
-w	Não desvincula arquivos temporários.
-e	Inclui o *flush* e faz a escrita ser síncrona.

Fonte: Elaborado pelo autor.

A Figura 11 exibe uma possível saída do *software* nas distribuições Linux.

Figura 11 – IOzone teste

```
Iozone: Performance Test of File I/O
        Version $Revision: 3.429 $
        Compiled for 64 bit mode.
        Build: linux-AMD64

Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
             Al Slater, Scott Rhine, Mike Wisner, Ken Goss
             Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
             Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
             Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
             Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
             Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
             Vangel Bojaxhi, Ben England, Vikentsi Lapa.

Run began: Thu Oct  4 20:01:59 2018

O_DIRECT feature enabled
File size set to 10240 kB
Record Size 4096 kB
Setting no_unlink
Include fsync in write timing
Command line used: iozone -i 0 -i 1 -i 2 -I -s 10M -r 4096 -w -e
Output is in kBytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 kBytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.
                                                            random    random
      kB  reclen    write  rewrite    read    reread    read     write
   10240    4096    13154    11405     4937    38950    39981     3358
```

Fonte: Elaborado pelo autor.

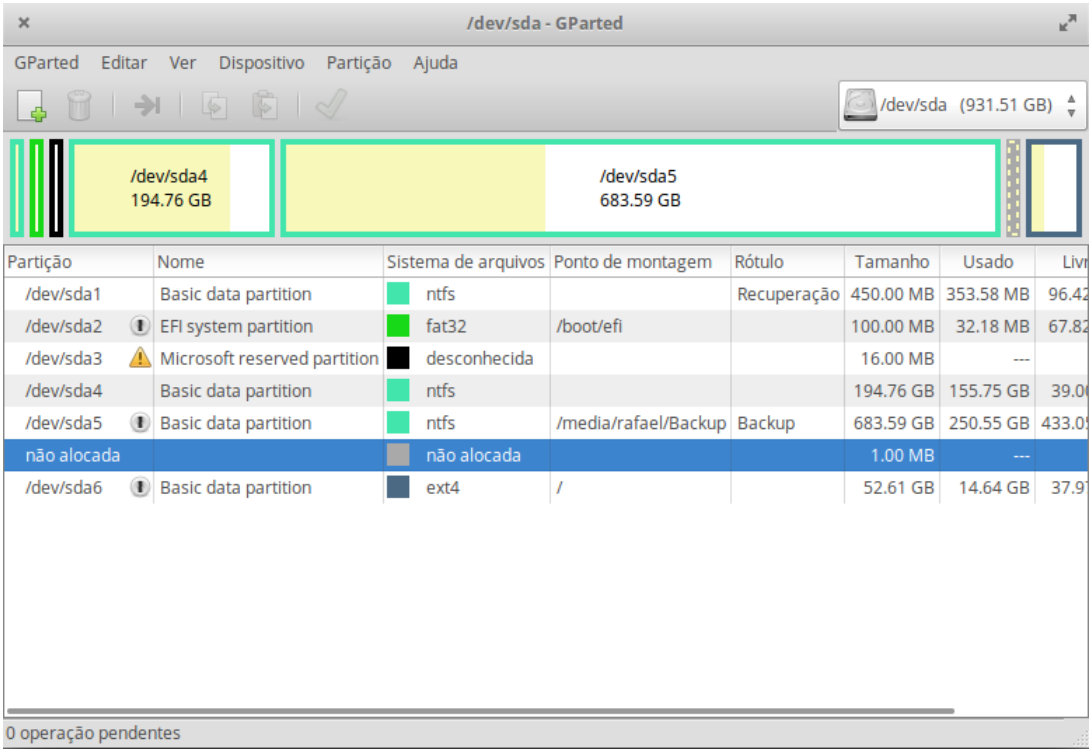
[1] <http://www.iozone.org/>. Acessado em mar/19

Para o nosso trabalho os campos *rewrite* e *reread* não foram utilizados, pois essas operações são relacionadas, respectivamente, a uma escrita e leitura usando um *cache buffer*.

3.1.2.2 Gparted

O Gparted[1] é um editor de partições gratuito que tem como finalidade o gerenciamento do disco via uma API simples e de manipulação gráfica, dando o usuário mais segurança ao realizar operações de formatação do disco. A Figura 12 ilustra o *software* em questão, com uma visão da sua interface principal.

Figura 12 – Software Gparted



Fonte: Elaborado pelo autor.

3.1.2.3 R Project

O R[2] é um *software* gratuito que conta com um ambiente para computação estatística e gráfica. Este programa teve bastante relevância, pois é com ele que levantamos as análises e estudos sobre os resultados presente na seção 5.

[1] <https://gparted.org/>. Acessado em mar/19
[2] <https://www.r-project.org/>. Acessado em mar/19

3.1.2.4 Gnuplot

O gnuplot[1] é um utilitário de criação de gráficos orientado por linha de comando para as distribuições Linux. Esse *software* permite gerar saídas de gráficos na tela bem como em inúmeros formatos, como PNG, EPS, SVG, JPEG. O código-fonte é protegido por direitos autorais, mas distribuído gratuitamente.

3.1.2.5 NetBeans IDE 8.2

Criado pela Sun Microsystems, O NetBeans[2] é um projeto de código aberto dedicado ao fornecimento de produtos sólidos de desenvolvimento de *software* (o NetBeans IDE e a Plataforma NetBean).

3.2 Métodos (Metodologia)

O conteúdo presente nesta subseção refere-se a toda revisão da literatura metodológica (Bibliométrica e Bibliográfica) usada para a realização do TCC, bem como alguns detalhes do projeto e ajustes que se fizeram necessários.

3.2.1 Bibliométrica

A bibliometria, segundo conta Lira (2010) é um método de pesquisa que permite encontrar uma quantidade restrita de periódicos essenciais (denominados nucleares) que se supõe possuir os artigos mais relevantes publicados sobre um determinado assunto, "partindo da prática estabelecida na comunidade científica de fornecer as referências bibliográficas de qualquer trabalho".

Seguindo o âmbito do conceito da revisão bibliométrica, inicialmente foram levantados diversos artigos que abordam o tema deste TCC nas principais bases de dados de artigos científicos, tais como: *ACM Digital Library, IEEE Xplore Digital Library, Elsevier's Scopus* e *Springer Link*, disponibilizados pelo site principal da CAPES[3] (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior).

Para as pesquisas nas bases de dados citados anteriormente foram levados em consideração toda a fase do planejamento da metodologia PDCA, que ainda será citado nesta seção. Tendo em mãos todo o conteúdo que será abordado, buscou-se pelos principais temas nas referidas bases.

A seleção dos artigos se obteve por meio de filtros disponibilizados pelos periódicos. Dentro do processo de filtragem foram consideradas palavras chaves, quantidade de citações

[1] <http://www.gnuplot.info/>. Acessado em mar/19
[2] <https://netbeans.org/features/java/index_pt_BR.html>. Acessado em mar/19
[3] <https://www.capes.gov.br/>. Acessado em mar/19

que determinado artigo conseguiu atingir e a relevância do autor (quantidade de artigos publicados).

3.2.2 Bibliográfica

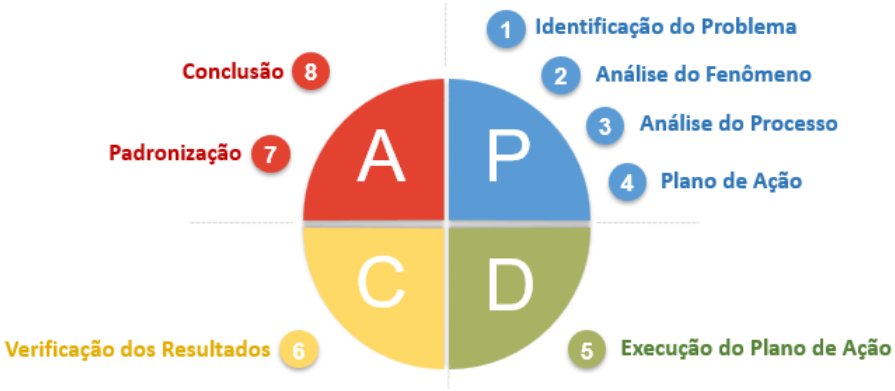
Do subconjunto de referências bibliográficas levantada na etapa anterior, cada um dos trabalhos foram analisados pelo seus resumo/*abstract*. Após avaliado a relevância do seu conteúdo, alguns dos artigos selecionados foram resenhados e até descartados conforme o assunto, pois muitos dos artigos lidos não tangia o objetivo principal deste TCC.

3.2.2.1 PDCA (*Plan-Do-Check-Ack*)

O Ciclo PDCA, é uma ferramenta de gestão de quatro passos que visa melhorar e controlar os processos e produtos de forma contínua. Tem por princípio tornar mais claros e ágeis os processos envolvidos na execução da gestão, identificando as causas dos problemas e as soluções para os mesmos e pode ser utilizado em qualquer organização de forma a garantir o sucesso nos negócios, independentemente da área ou departamento visado (ANDRADE, 2015).

A Figura 13 demonstra e explica cada parte do processo.

Figura 13 – Metodologia PDCA



Fonte: (COUTINHO, 2017)

3.2.3 Projeto e Implementação

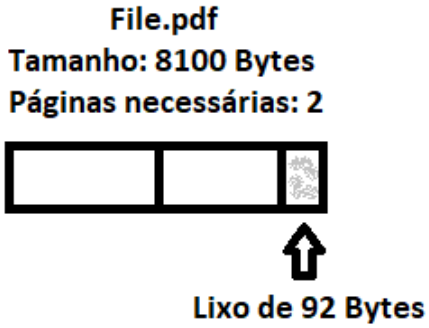
Inicialmente estudamos quais foram as estratégias usadas pelos principais FS existentes e consolidados (EXT4, BTRFS e NTFS). Depois de estudados e entendidos, dividimos o projeto do FS nas principais estruturas de responsabilidade do Superbloco. Para isto utilizamos a abstração de um sistema de arquivo apresentada pelo Silberschatz (2009) (Ver seção 2, subseção 2.1).

As estruturas de dados e algoritmos foram implementadas de maneira personalizadas pelo próprio autor na linguagem de programação C, não sendo utilizada biblioteca de TADs para as operações que manipulam a aplicação proposta. As únicas bibliotecas utilizadas foram de carácter operacional do sistema, como: *<stdio.h>, <unistd.h>, <string.h>, <stdlib.h>, <string.h>, <assert.h>, <math.h>, <time.h>*, elas sendo necessárias para a execução da aplicação.

3.2.4 Verificação e Ajustes

Inicialmente o sistema de arquivos foi testado como um arquivo binário representando, volume virtual no qual eram colocados e recuperados exemplos de mídias (ex: pdf, mp3, csv) e, após a inserção, era verificada a integridade do mesmo. Após investigarmos o conteúdo de mídia recuperada do FS implementado neste trabalho, observamos a existência de *bytes* nulos no final do arquivo, devido ao modo como era inicialmente fora resgatado o conteúdo da mídia no FS. Tal observação alertou a necessidade de tratar a fragmentação interna na última página do arquivo, conforme ilustra a Figura 14.

Figura 14 – Fragmentação interna de arquivo de 8,1 KB em páginas de 4 KiB



Fonte: Elaborado pelo autor.

Para solucionar o problema descrito acima, bastou consultar o tamanho total do arquivo na estrutura de seu *inode* e retirar o resto da diferença entre o tamanho total das páginas recuperadas e o tamanho real do arquivo. Assim, conseguiu-se obter os *bytes* do arquivo sem o "lixo" gerado pela fragmentação interna na última página. O pseudocódigo apresentado no Algoritmo 1) define tal processo.

Algorithm 1 Leitura no FS

```
1: procedure LERDOFS(total_paginas, tamanho_arq, path_file)
2:     aux = 0
3:     buffer_arquivo = 0
4:     quantidade_lida = 0
5:     diff = 0
6:     File ← open(path_file)
7:     while (aux! = total_paginas − 1) do          ▷ leia até a penultima página
8:         buffer_arquivo ← File
9:         quantidade_lida = ftell(File)
10:        aux + +
11:    diff ←tamanho_arq - quantidade_lida                          ▷ Diferença
12:    buffer_arquivo ← File                    ▷ File está com a diferença (diff)
13:    return(buffer_arquivo)
```

Fonte: Elaborado pelo autor.

Também, foram realizadas medidas do tempo gasto para colocar uma mídia dentro do FS, bem como para recuperá-la, visando ter uma noção do desempenho do FS ao longo da implementação. Ao identificar problemas de desempenho nas operações de escrita, revisamos o código do FS implementado para identificar possíveis otimizações, tais como economizar na quantidade de chamada de sistemas realizadas. Um outro exemplo de otimização de estrutura foi quando notamos que nossa abordagem de manipulação dos blocos livres estava demandando muitas operações, aumentando significativamente a complexidade do algoritmo proposto, o que era de fato algo desnecessário.

Anteriormente estávamos usando uma abordagem dinâmica de alocação, ou seja, quando um arquivo precisava ser alocado, nossa estrutura verificava as páginas que não foram usadas e então alocava-as, salvando-as na estrutura do superbloco em tempo de execução.

A justificativa para esta abordagem é que de alguma forma economizaríamos, por um instante, em gasto de espaço. O superbloco usando esta tática dinâmica não ficaria grande, gastando pouco espaço de memória, uma vez que sempre é salvo no início da nossa memória disponibilizada, a cada operação feita, garantindo a sincronização e a persistência dos dados salvos. A Figura 15 tem como finalidade apresentar uma ocorrência do problema identificado.

Figura 15 – Ilustração do problema identificado

Fonte: Elaborado pelo autor.

Contrapondo a justificativa para a alocação dinâmica dos blocos livres, percebemos que a nossa estrutura teria uma ordem de complexidade de *O(n)*, pois para verificarmos suas páginas livres, teríamos de percorrer toda a estrutura, para assim então salvar um outro arquivo.

Um outro argumento que refutou a abordagem de alocação dinâmica destes blocos foi que, em alguma hora, o FS poderia ficar cheio, ou seja, com todo o espaço disponível ocupado. Assim todas as páginas iriam ser de algum modo alocadas no superbloco, fazendo com que o superbloco tenha o maior espaço disponível na memória disponibilizada.

Já que percebemos que em algumas situações o FS teria que disponibilizar todas as páginas para a consulta de disponibilidade, pensou-se que o melhor jeito de tratar este problema seria, ao criar o superbloco que toda a estrutura de blocos livres já deveria estar alocada no mesmo.

Ao optar por esta abordagem poderíamos acessar qualquer página, na hora que fosse demandado, em nível de complexidade O(1), pois agora em nossa estrutura as páginas estavam em ordem, já previamente alocadas de forma que para acessá-las bastaria fazer uma simples conta de deslocamento. Tal conta, é dada por:

$$\Delta = \tau * \kappa \tag{3.1}$$

Onde Δ é a página em questão, τ é uma constante que diz o tamanho de cada pedaço da estrutura de blocos, e κ é o número da página que deve ser acessada.

Feito isso notamos um ganho satisfatório no desempenho do nosso FS, então logo acatamos esta ideia e fizemos os devidos ajustes nas estruturas para adaptação da nova forma de manipulação.

4 DESENVOLVIMENTO

Neste tópico será abordado todo o desenvolvimento do projeto e detalhes da sua implementação.

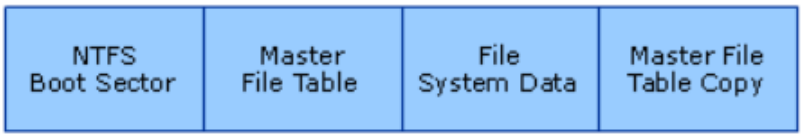
4.1 Estrutura dos FS mais populares

Para sabermos o real motivo de determinado FS se sobressair em relação a outro (e.g., maior vazão aferida na leitura e na escrita, aleatória e/ou sequencial), é necessário estudar as estruturas de dados e estratégias utilizadas por cada sistema de arquivos. Este tópico do trabalho tem como finalidade demonstrar como são as estruturas dos FS apresentados na fundamentação teórica (seção 2 deste documento).

4.1.1 NTFS (vide seção 2.6)

Relembrando o que foi abordado na seção 2.6 deste artigo, o NTFS contém um arquivo principal (superbloco), denominado *Master File Table*, que tem como finalidade gerenciar todos os seus metadados, ou seja, gerenciar e armazenar toda as operações feitas no FS em questão (Ver seção 2.1). De forma geral, previamente, é necessário ter uma visão macro sobre as principais estruturas que iniciam este FS. A Figura 16 tem como finalidade demonstrar a estrutura organizacional do NTFS (MICROSOFT, 2009).

Figura 16 – Estrutura organizacional do NTFS.



Sintetizando cada um dos campos que foram mostrados na Figura 16, de acordo com a documentação do NTFS provida no site do sistema operacional Windows[1], os tópicos abaixo expressam o que cada parte significa.

1. ***NTFS Boot Sector*** - Contém o bloco de parâmetros da BIOS (*Basic Input / Output System*) que armazena informações sobre o *layout* do volume e as estruturas do sistema de arquivos, bem como todo o código de inicialização

2. ***Master File Table*** - Contém as informações necessárias para recuperar arquivos da partição NTFS, como os atributos de um arquivo.

[1] <https://docs.microsoft.com/en-us/previous-versions/windows/>. Acessado em mai/19

3. ***File System Data*** - Armazena todos os dados que não estão contidos na tabela de arquivos mestre, ou seja, arquivos em geral.

4. ***Master File Table Copy*** - Inclui cópias dos registros essenciais para a recuperação do sistema de arquivos, se houver um problema com a cópia original.

Posteriormente de compreensão da estrutura organizacional do NTFS, pudemos entender como ele lida com o armazenamento de seus dados. Como é descrito na documentação do NTFS, todos os FS do Windows são organizados baseado em *clusters*. Um *cluster* (ou unidade de alocação) simplesmente é a menor de quantidade de espaço em disco que pode ser alocada para armazenar um arquivo. Os valores do tamanho do *cluster* podem variar, pois sua dimensão é determinada pelo número de setores (unidades de armazenamento em um disco rígido). Por exemplo, em um disco que usa setores de 512 bytes, um *cluster* de 512 bytes contém um setor, enquanto um *cluster* de 4 kilobytes (KB) contém 8 setores.

Os *clusters* em um dispositivo de memória secundária formatada com o NTFS são numerados sequencialmente desde o início da partição em números de *cluster* lógicos. O NTFS armazena todos os objetos do sistema de arquivos usando a MFT (*Master File Table*), semelhante em estrutura a um banco de dados.

Conforme conta em sua documentação. Como o NTFS usa tamanhos de *cluster* diferentes, dependendo do tamanho do volume, cada sistema de arquivos tem um número máximo de *clusters* que ele pode suportar. Quanto menor o tamanho do *cluster*, mais eficientemente um disco se torna em armazenar informações, pois o espaço não utilizado em um *cluster* não pode ser usado por outros arquivos. E quanto mais *clusters* um sistema de arquivos suportar, maiores serão os volumes que você pode criar e formatar usando um determinado sistema de arquivos. O NTFS usa tamanhos de *cluster* menores, o que torna uma estrutura de organização de arquivos mais eficiente. O tamanho padrão que este FS aloca para os *clusters* são descritos na Tabela 5, que tem a finalidade de demonstrar as faixas de valores dos *clusters* em relação ao tamanho do volume do disco.

Tabela 5 – Relação tamanho do disco e clusters NTFS.

Tamanho do disco	Tamanho dos clusters no NTFS
7 megabytes (MB) - 512MB	512 bytes
513 MB - 1,024 MB	1KB
1,025 MB - 2GB	2 KB
2GB - 2 Terabytes	4 KB

Fonte: (MICROSOFT, 2009)

Toda operação que altera um dado de um arquivo é salva na MTF, e o conteúdo do respectivo dado é salvo nos *clusters*, que corresponde ao espaço disponível na MTF.

4.1.2 EXT4 (vide seção 2.4)

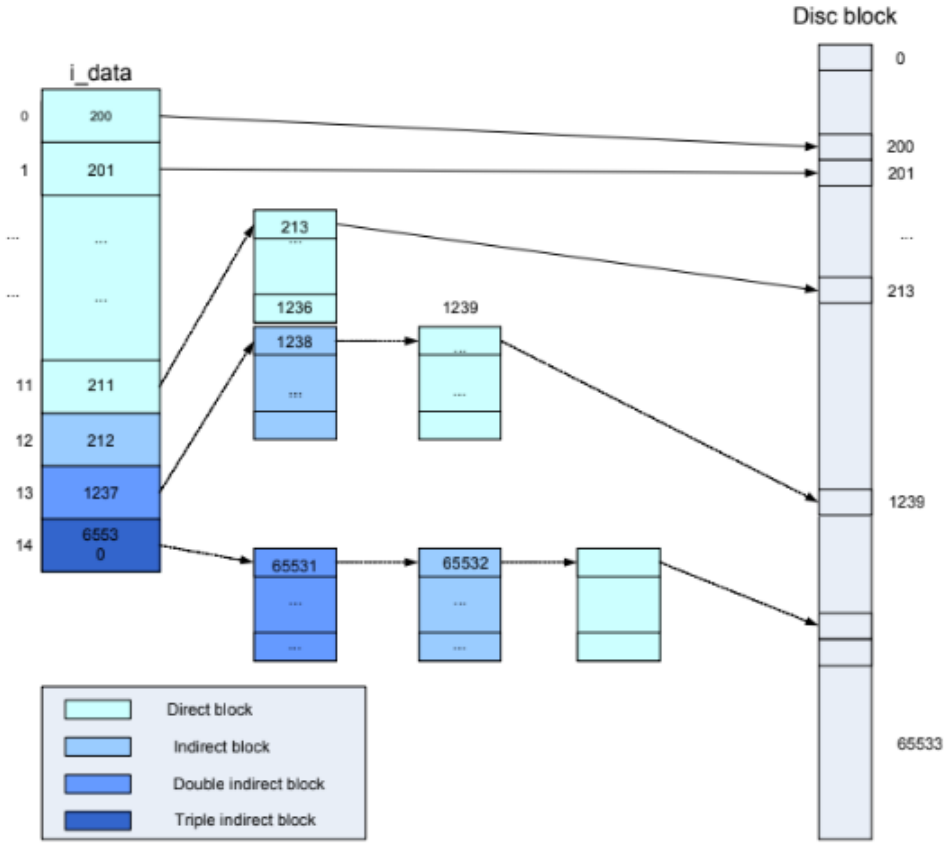
Para explicar a anatomia do EXT4, primeiramente temos que expressar uma das principais desvantagens presente em seu antecessor, o EXT3, que era basicamente no seu método de alocação. Segundo Jones (2009) os arquivos eram alocados usando um tipo de mapa de bits de espaço livre, que não era muito rápido e tão pouco escalável. Jones também explica que o formato EXT3 é muito eficiente para arquivos pequenos, mas péssimo para arquivos grandes. Portanto, o EXT4, substituiu o mecanismo de gerenciamento de blocos livres do EXT3 por "extensões" com a finalidade de melhorar o desempenho na alocação e uma estrutura de armazenamento.

> "Uma extensão é basicamente uma maneira de representar uma sequência contígua de blocos. Fazendo isso, os metadados presentes no EXT4 tendem a reduzir pois, em vez de manter informações sobre onde todo e cada bloco é armazenado, a extensão mantém informações sobre onde uma longa lista de blocos contíguos é armazenada reduzindo, assim, a demanda por armazenamento geral de metadados." (JONES, 2009).

Complementarmente, Borislav Djordjevic; Valentina (2012) esclarece que o sistema de arquivos EXT4, é baseado em uma alocação dinâmica dos *inodes*, fornecendo um bom desempenho ao sistema, bem como robustez e compatibilidade. Para poder manipular com sucesso diferentes tamanhos de arquivos, o EXT4 implementa dinamicamente *inodes* de 64 bits. Além disso, ao contrário do EXT3 que usa blocos de mapeamentos indiretos (Figura 17), o EXT4 independentemente do tamanho do arquivo garante uma alocação de arquivos mais eficiente com base em um alocador de bloco especial e estratégias para diferentes requisitos de alocação. Uma das estratégias do referido FS é assegurar que os arquivos pequenos serão armazenados continuamente, um após o outro (Figura 18).

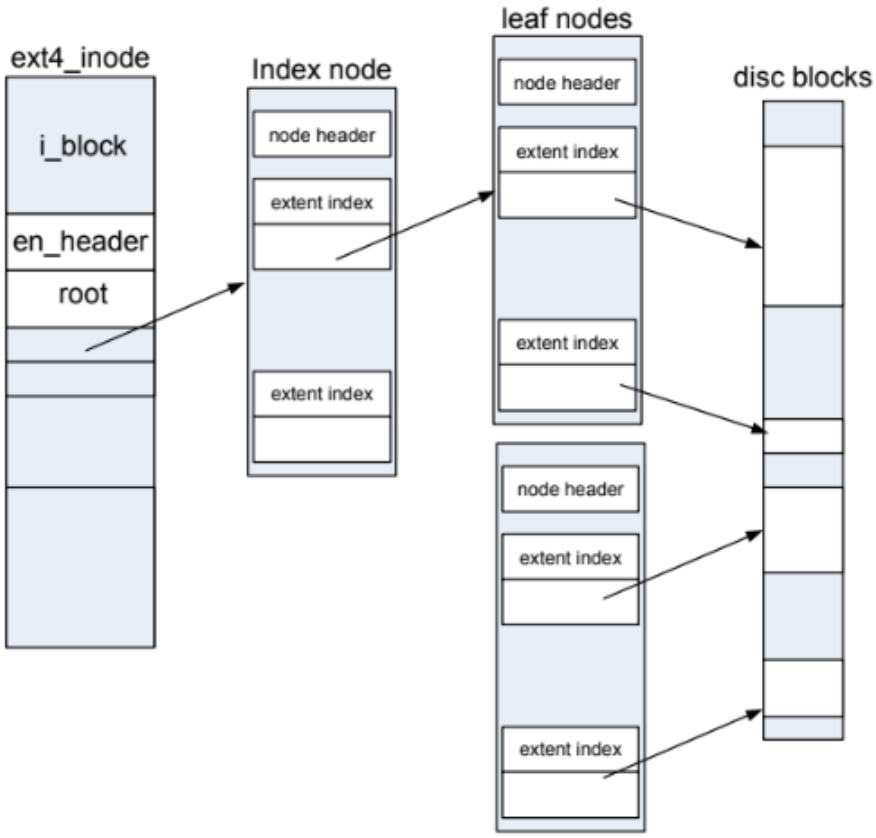
As Figuras 17 e 18 tem como finalidade demonstrar como é feito um acesso a um dado na estrutura, respectivamente, do EXT3 e EXT4.

Figura 17 – Estrutura EXT3.



Fonte: (BORISLAV DJORDJEVIC; VALENTINA, 2012)

Figura 18 – Estrutura EXT4.



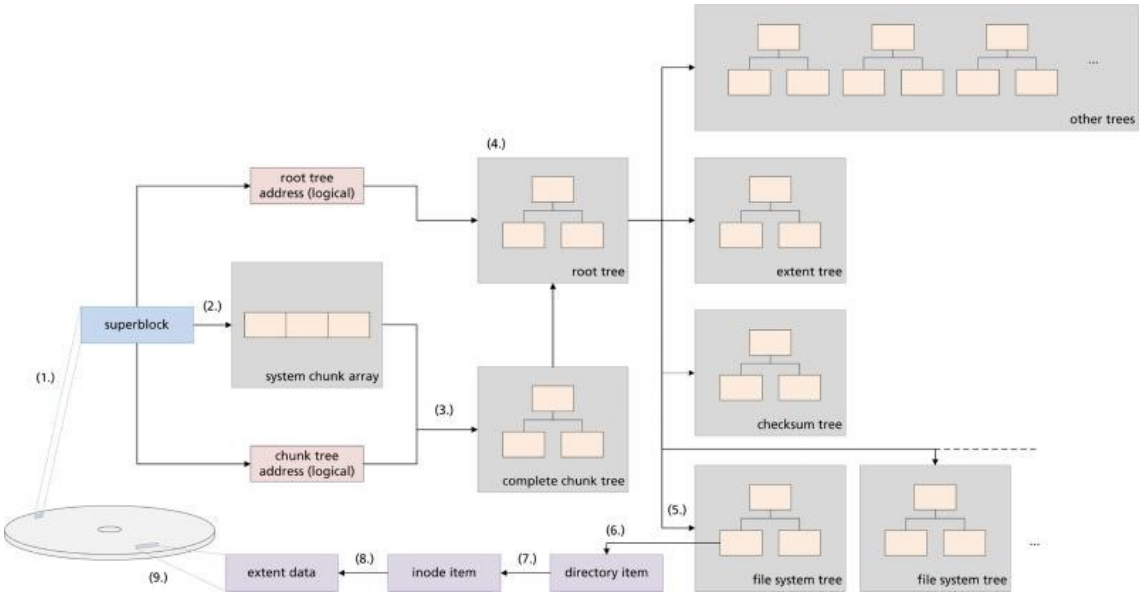
Fonte: (BORISLAV DJORDJEVIC; VALENTINA, 2012)

4.1.3 BTRFS (vide seção 2.5)

Sobre o BTRFS, é importante entender como são estruturados seus blocos e o percurso para acessar um arquivo em sua estrutura de dados, ilustrado na Figura 19. A Figura 19 ilustra o passo a passo citado abaixo de tal processo que, segundo Jan-Niclas Hilgert; Martin (2018), contam:

1. A primeira instância diz respeito à localidade do superbloco no disco, que se encontra no seu endereço físico padrão 0x10000;
2. São extraídos os itens do "fragmento do sistema", armazenados no superbloco para o mapeamento lógico inicial do endereço físico;
3. É localizado o endereço lógico da árvore de fragmentos no superbloco, traduzindo-o para sua parte física e construindo a árvore de fragmentos. De agora em diante, essa árvore será usada para realizar o mapeamento de endereços lógicos para físicos;
4. Então, localiza-se o endereço lógico da árvore raiz no superbloco, traduzindo-o para o endereçamento físico e construindo a árvore raíz;
5. A árvore raiz armazena os endereços lógicos das raízes das outras árvores, incluindo as próprias árvores do sistema de arquivos. Logo após, é localizado o endereço da raíz da árvore do sistema de arquivos correspondente, traduzindo e construindo outra árvore;
6. Por conseguinte, busca-se nessa árvore de sistema de arquivos para localizar o arquivo de interesse. Seu nome é armazenado em um item de diretório;
7. É realizada a leitura do *inode* correspondente ao arquivo desejado, para recuperar seu identificador e metadados;
8. Utiliza-se o identificador como chave, para encontrar seus itens na árvore do sistema de arquivo. Ou seja, faz-se a conversão de um caminho virtual para o caminho absoluto;
9. Finalmente, são extraídos os dados descritos por todas as extensões correspondentes ao arquivo, mapeando seus endereços lógicos em físicos.

Figura 19 – Visão geral das estruturas do BTRFS mais importantes para o acesso a um arquivo.



Fonte: (JAN-NICLAS HILGERT; MARTIN, 2018)

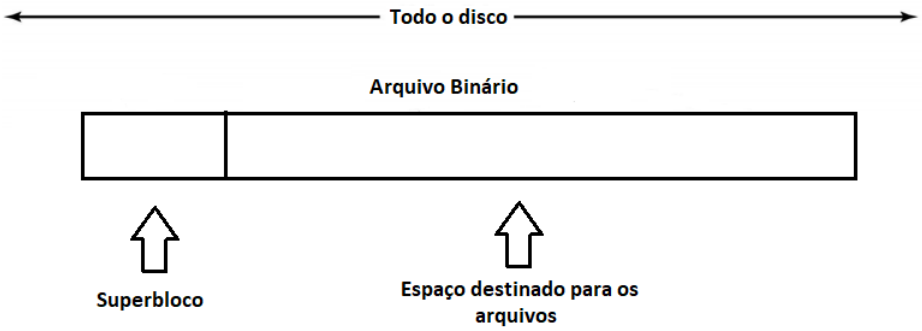
4.2 Estruturas do FS proposto

Após compreender como trabalham os FS citados na fundamentação teórica (Ver seção 2), foi tirado algumas ideais que poderiam complementar para a criação do FS que este trabalho desenvolve. Este presente tópico tem como finalidade demonstrar como são as principais estruturas do FS desenvolvido.

4.2.1 Metadados do Sistema de Arquivo (*Volume Control Block*)

O *Volume Control Block* ou Superbloco é onde ficam armazenadas todas as estruturas para as operações que controlam um sistema de arquivos (Ver seção 2, subseção 2.1). Em nosso projeto, o superbloco sempre é escrito no começo de um arquivo binário, a fim de emular uma situação real. A Figura 20 tem como finalidade demonstrar como é feito o processo descrito.

Figura 20 – Processo de escrita do Superbloco



Fonte: Elaborado pelo autor.

O nosso superbloco é um registro de dados heterogêneos, cujos metadados necessários para as operações de um FS são descritos na Tabela 6.

Tabela 6 – Metadados do superbloco.

Campo	Descrição
Max_size_file	Define o tamanho máximo do maior arquivo que o FS suporta.
Size_memory	Comporta o tamanho do volume de todo o FS.
Free_space_bytes	Comporta a quantidade de espaço livre do FS em *bytes*.
Filled_space_bytes	Comporta a quantidade de *bytes* já preenchidos no FS.
Begin_pagingList	Comporta o início em *bytes* de onde começa a estrutura de gerenciamento de blocos livres.
Begin_inodeList	Comporta o início em *bytes* de onde começa a estrutura dos *inodes*.
End_inodeList	Comporta onde é o fim, em *bytes*, da estrutura dos *inodes*.
Max_inode	Quantidade máxima de *inodes* que nosso FS suporta.
Paging_List	Ponteiro para a estrutura que gerencia os blocos livres.

Fonte: Elaborado pelo autor.

O superbloco em questão bem como todo o FS é escrito/lido de maneira serializada com o deslocamento de 4096 *bytes* (4KiB que é o *cluster* padrão utilizado na maioria dos dispositivos de memória com acesso aleatório), através de funções implementadas pelo próprio autor.

Diferentemente dos FS que usam a tecnologia COW (Ver seção 2.5), a cada operação feita no nosso sistema de arquivos o superbloco é reescrito, também no início do arquivo,

a fim de manter a integridade dos dados.

4.2.2 Metadados do Arquivo (*File Control Block*)

O *File Control Block* ou como conhecido nas distribuições Linux como *inode*, são onde ficam armazenadas todas as informações dos arquivos, bem como o seu caminho absoluto. (Ver seção 2, subseção 2.1).

A implementação do *inode* foi feita com um registro e contém os seguintes metadados descritos na Tabela 7.

Tabela 7 – Metadados do inode.

Campo	**Descrição**
Path	Comporta o nome do caminho absoluto de onde está o arquivo no nosso FS.
File_name	Comporta o nome do arquivo.
File_size	Comporta o tamanho que este determinado arquivo contempla.
Flag	Campo que distingue se o *inode* em questão é um arquivo ou uma pasta.
Time	Comporta a data de criação do *inode*.
Ender_List	Comporta a lista de endereço das páginas onde o arquivo foi salvo.

Fonte: Elaborado pelo autor.

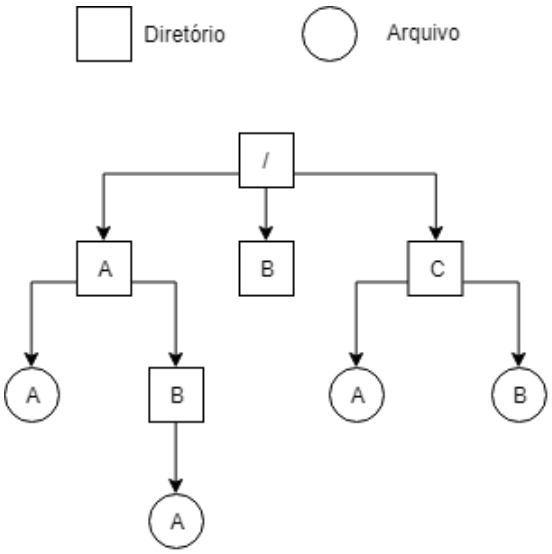
4.3 Estruturas de Diretórios

Uma estrutura de diretórios tem como finalidade organizar e mostrar ao usuário como os seus dados estão sendo salvos em um FS, dando uma ideia abstrata de níveis e subníveis de um arquivo ou pasta dentro de um sistema de arquivos.

Nossa estrutura de diretório é um registro que comporta a ideia de uma estrutura do tipo árvore, seguindo o padrão Linux: o nosso *root directory* ou diretório raiz começa com o nome absoluto /.

Nossa árvore de diretórios mapeia um determinado caminho absoluto de um arquivo ou diretório para seu respectivo *iNode* a fim de organizar o acesso. Como pastas e arquivos compartilham da mesma estrutura *iNode*, a quantidade máxima de pastas e arquivos fica limitado pelo campo *Max_inode* que representa uma constante de valor 976.562 presente no superbloco, a explicação da escolha deste valor é que este é o valor do tamanho do dispositivo dividido pelo valor de cada página (4096 bytes). A Figura 21 tem como finalidade demonstrar como é essa estrutura dentro do FS em questão.

Figura 21 – Dentry criado.



Fonte: Elaborado pelo autor.

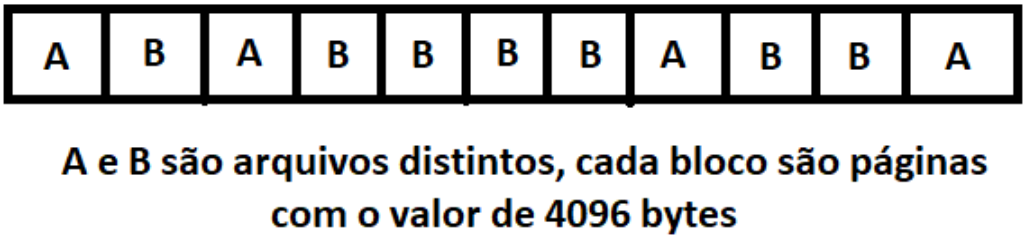
4.3.1 Alocação de Blocos

Como foi apresentado em sua tabela na seção 4.2.2, campo *Ender_List*, os *iNodes* contém um vetor indexado com os números das páginas que contém os dados do arquivo. A quantidade máxima de páginas que um arquivo pode ter pode ser calculada como o tamanho do *Max_size_File* (Ver nessa Tabela 6 desta seção, subseção 4.2.1) dividido por 4096 *bytes* (valor correspondente a uma página), o valor dessa operação nos dá a quantidade de máximas páginas.

Seguindo em nossas funções, para a alocação de um arquivo é necessário previamente verificar a estrutura que gerencia o espaço livre a fim de saber se existe espaço suficiente para o seu armazenamento, só após verificado invocamos uma função que nós retorna um vetor com os índices das relativas páginas livres, fazendo assim a escrita com o deslocamento recebido.

Esta tática de alocar os dados em páginas de 4096 bytes é uma boa estratégia, pois este espaço em questão remete a ideia de uma célula de memoria *Flash*. Como todos os acessos em mídias que usam esta tecnologia *Flash* podem ser realizados via acesso aleatório, não temos a perda de desempenho ao deslocarmos entre células não contíguas. A Figura 22 demonstra uma visão abstrata de 2 arquivos salvos em nossa estrutura com a finalidade de ilustrar a Alocação dos blocos.

Figura 22 – Alocação dos blocos

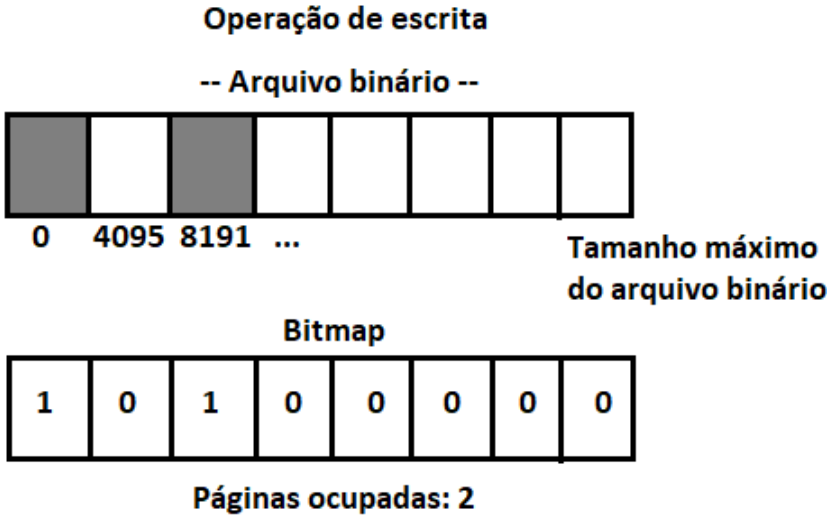


Fonte: Elaborado pelo autor.

4.3.2 Gerenciamento do Espaço Livre

Uma estrutura *Bitmap* controla se determinada página está sendo utilizada ou não, o seu valor 0 indica que uma página está livre e o valor 1 que está ocupada. Além deste campo a estrutura comporta o número da página bem como o valor em *bytes* do seu início e o seu fim, estes que são necessários apenas para uma visualização dos arquivos na mídia. O gerenciamento do espaço livre é sempre consultado quando necessário. A Figura 23 abaixo exemplifica tal gerenciamento.

Figura 23 – Operação de escrita do FS desenvolvido.



Fonte: Elaborado pelo autor.

4.3.3 VFS

As assinaturas das funções de execução das operações foram inspirada no VFS (linux/fs.h) visando expor a mesma API, para que posteriormente possa ser acoplada em nível Kernel do Linux.

Observe-se que nem todas as funções previstas no VFS foram implementadas por não serem necessárias para a análise de desempenho dos sistemas de arquivos selecionados nos dispositivos de armazenamento alvo. Também consideramos a limitação de tempo para a realização de um TCC.

4.3.4 FUSE e montagem

Realizamos o mapeamento de operações do nosso FS para a API FUSE (libfuse/fuse.h) de maneira que consegue-se utilizar o sistema de arquivos no espaço de memória do usuário. Porém o FUSE não permite manipular/salvar as configurações feitas no superbloco na mídia em questão, ou seja, o sistema de arquivos proposto seria manipulado completamente na memória principal, tanto para o uso das páginas bem como toda nossa estrutura, fazendo com que abandonássemos tal abordagem durante a implementação do trabalho proposto.

4.4 Formatação do FS

Antes de poder realizar operações no nosso FS é necessária uma formatação, que é a escrita do superbloco em um arquivo binário, de maneira serializada. O próprio superbloco fica escrito nas primeiras χ páginas que são bloqueadas contra acessos do usuário. O valor de χ inicial é obtido pela fórmula:

χ= metadados do superbloco + estrutura *bitmap* + estrutura da árvore de diretórios

Todos os valores calculados em χ estão em bytes, pré-definidos pelo tamanho do arquivo binário em questão.

5 RESULTADOS E ANÁLISE

Nesta seção serão apresentados os experimentos realizados, os resultados obtidos e suas análises. Em todos os testes, foi utilizada a ferramenta de *benchmark* IOzone, para aferir o desempenho de alguns dos sistemas de arquivos (FS) populares e promissores, em diferentes tecnologias de armazenamento persistente.

5.1 Descrição dos experimentos

Para realizar os testes de desempenho com o IOzone, basta antes formatar o dispositivo de armazenamento com o sistema de arquivo alvo (e.g., NTFS, EXT4, BTRFS) e, então, executar o software IOzone com os parâmetros de teste desejados (vide seção 3.1.2.1). Os experimentos para aferir o desempenho dos sistemas de arquivos foram conduzidos em três dispositivos de armazenamento, sendo eles dois com tecnologias de acesso aleatório e um com acesso sequencial: *Flashdrive* ("*Pendrive*"), *Solid State Drive* (SSD) e *Hard Disk* Drive (HDD) . Os resultados das medições serão apresentados na seção 5.2.

No caso do protótipo de sistema de arquivos desenvolvido neste TCC, devido ao prazo e às limitações técnicas (integrar o FS ao Kernel Linux), optamos por verificar o desempenho das operações do nosso sistema de arquivos sobre um FS subjacente, tradicional. Observe que, para formatar um dispositivo físico de armazenamento com um FS personalizado e efetivamente "montá-lo" no sistema operacional, é necessário recompilar o Kernel Linux para incluir ao VFS (*Vitual File System*) o código do novo FS. O uso do IOzone foi pensado como base para avaliarmos o desempenho de FS populares em dispositivos de acesso aleatório. Assim, de posse do melhor desempenho possível para o dispositivo formatado com determinado FS, pôde-se inferir o *overhead* do novo FS desenvolvido ao comparar o desempenho da manipulação de um arquivo através do novo FS com o desempenho da manipulação diretamente no FS subjacente.

Para a realização dos testes de desempenho do FS desenvolvido, inicialmente foi realizada a formatação de um *pendrive* com o sistema de arquivos EXT4. Então, pré-alocamos no sistema EXT4 um arquivo de 4 GB para a emulação de um dispositivo de armazenamento. Por fim, esse arquivo binário é então formatado ao escrevermos no início dele o superbloco contendo as estruturas de dados do FS desenvolvido. Então, o arquivo de 4 GB formatado com o FS desenvolvido pode ser armazenado em um *Flashdrive*, HDD ou SSD (por sua vez gerenciados com o EXT4), para que sejam realizados os testes de escrita e leitura, tanto sequencial quanto aleatória.

5.2 Resultados e Análise

Este tópico do trabalho tem como finalidade apresentar os resultados bem como a análise dos dados obtidos.

5.2.1 IOZone3 (NTFS, EXT4, BTRFS)

Nesta subseção serão apresentados os resultados dos experimentos feitos nos respectivos FS, bem como suas comparações de desempenho. Os parâmetros utilizados no IOzone foram pensados para propor uma maneira de simular uma situação cotidiana no uso do FS proposto neste trabalho. Em tempo, vale ressaltar que foi padronizado para os sistemas de arquivo analisados o comprimento do registro (tamanho do menor bloco de dados) como sendo de 4096 *bytes*.

Podemos notar pelos testes executados nas diferentes mídias que com o *file size* de 10 MiB (arquivo pequeno), na maioria dos casos tem o seu resultado bastante divergente, ou seja, o seu resultado contém uma variação muito significativa. A explicação deste fator se dá pelos diversos ruídos gerados no início de uma gravação ou leitura, estes podendo ser *System calls* (chamadas de sistemas) em contexto de SO.

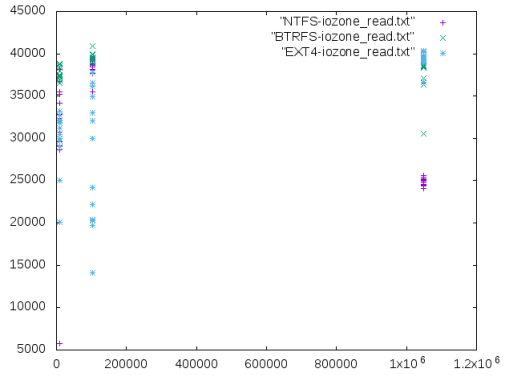
As Tabelas 8, 9 e 10 tem como finalidade comparar lado a lado as medidas retiradas na subseção anterior, mostrando para o leitor os comportamentos dos FS juntos, nas diferentes mídias, nota-se que em todas as tabelas que o eixo X corresponde ao tamanho do arquivo e eixo Y a vazão.

5.2.1.1 Desempenho dos FS com arquivo de 10, 100 e 1000 MiB (v2)

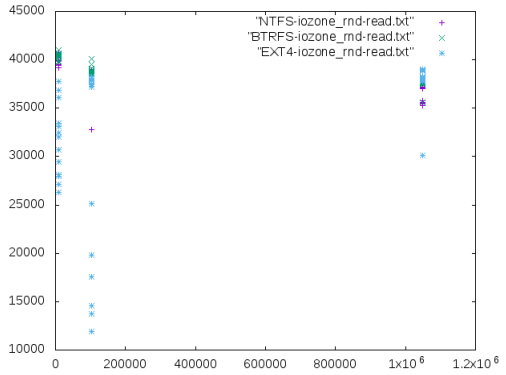
Tabela 8 – Comparação do desempenho de FS em um PenDrive

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

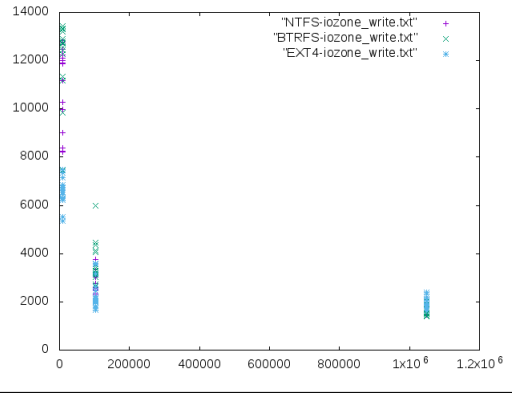
(a) Leitura sequencial



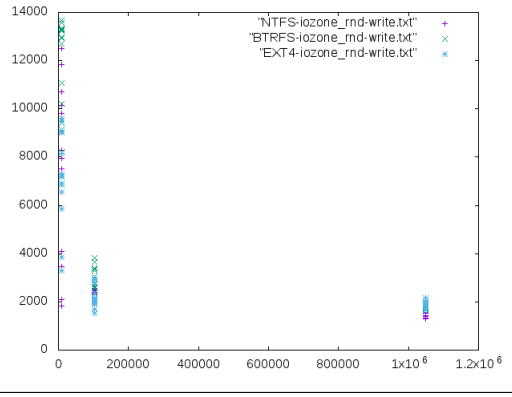
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória



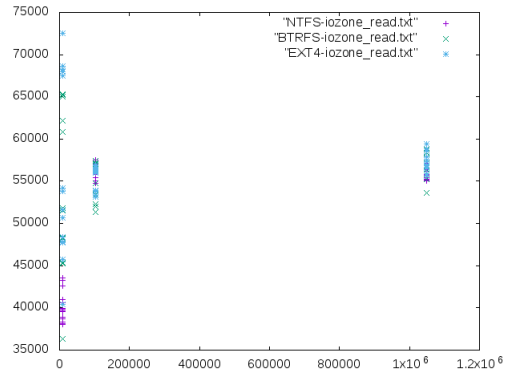
Fonte: Elaborado pelo autor.

No caso do dispositivo de armazenamento *Flashdrive*, observe como o sistema de arquivos NTFS obteve a pior vazão dentre os FS para arquivos de tamanho 1 GiB. Também, vale ressaltar que o BTRFS apresentou maior vazão que os demais FS para arquivos de tamanho 100 MiB.

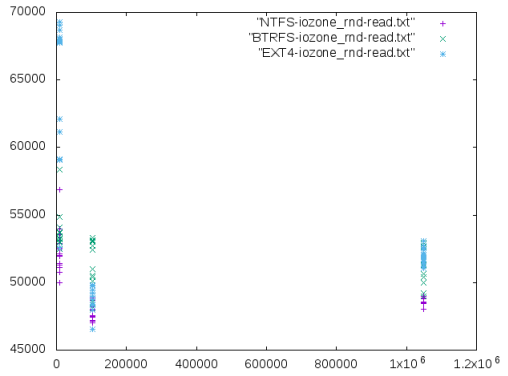
Tabela 9 – Comparação do desempenho de FS em um HDD

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

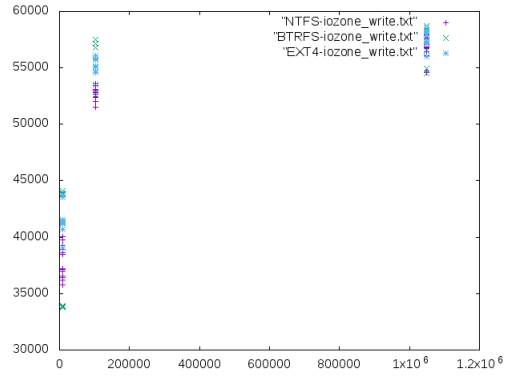
(a) Leitura sequencial



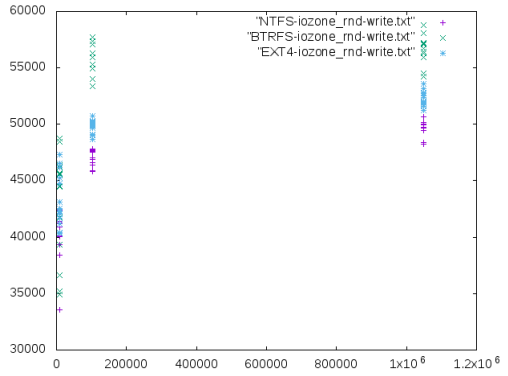
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória



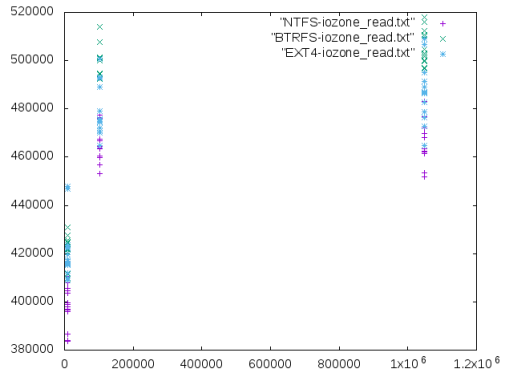
Fonte: Elaborado pelo autor.

Discorrendo os resultados obtidos na mídia de armazenamento HDD, observou-se que o sistema de arquivos NTFS obteve a pior vazão dentre os FS para operações de leitura ou escrita aleatórias. Também, vale ressaltar que o BTRFS apresentou maior vazão que os demais FS para tais operações não sequenciais.

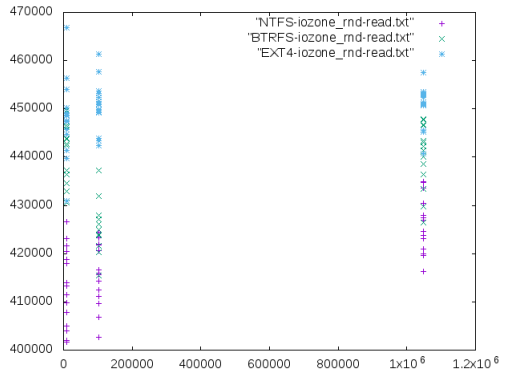
Tabela 10 – Comparação do desempenho de FS em um SSD

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

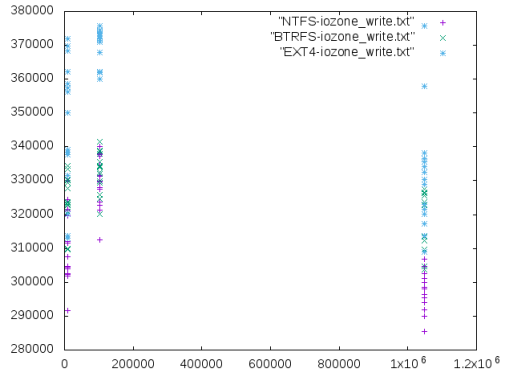
(a) Leitura sequencial



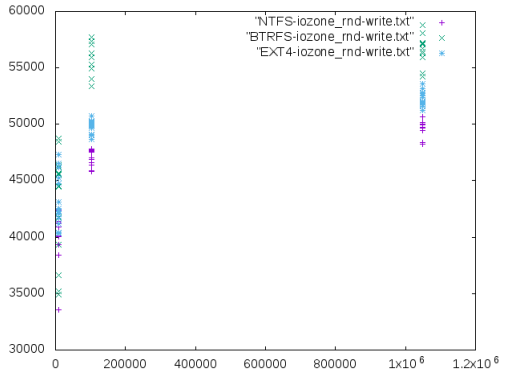
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória



Fonte: Elaborado pelo autor.

Já nas medições feitas no dispositivo armazenamento SSD, podemos observar que o sistema de arquivos BTRFS obteve a maior vazão dentre os FS para operações de leitura, enquanto que o EXT4 se destacou nas operações de escrita.

Tendo em vista que, após diversos testes realizados (ver anexo A), observamos que o desempenho dos FS fica mais estável (menor variabilidade) apenas com tamanhos de arquivo a partir de 214 MiB, analisamos e comparamos a mediana dos dados obtidos, com a finalidade de identificar o melhor desempenho entre os sistemas de arquivos presentes nessa seção. Para os experimentos não consideramos o arquivo de 214 MiB como sendo o maior tamanho a ser analisado, mas sim um de 1 GiB, pois assim nossos testes ficariam em uma escala de 10 vezes 1 MiB, 100 vezes 1 MiB e por fim 1000 vezes 1 MiB.

5.2.1.2 Desempenho dos FS com diferentes tecnologias de armazenamento

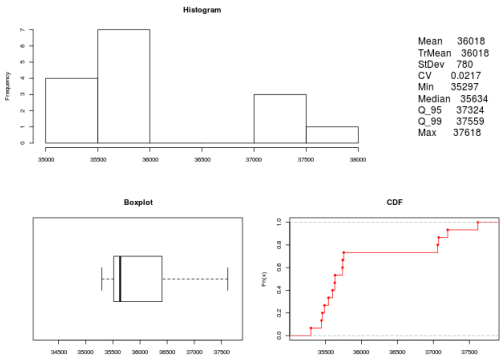
As Tabelas 11, 13 e 15 a seguir tem como finalidade apresentar um sumário estatístico das medições foram feitas, utilizando para tal um *script* codificado em R (Ver seção 3 materiais, subseção 3.1.2.3). Logo abaixo por questões de espaço e considerando

o cenário de pior caso, apresentaremos apenas as medições de escrita e leitura aleatória de cada FS e, posteriormente, uma comparação gráfica entre os sistemas de arquivos analisados.

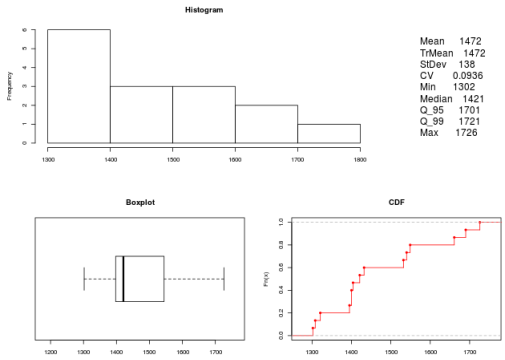
Tabela 11 – Vazão de R/W aleatória dos FS em um *Flashdrive*

IOzone *Benchmark* (*File size*: 1 GiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

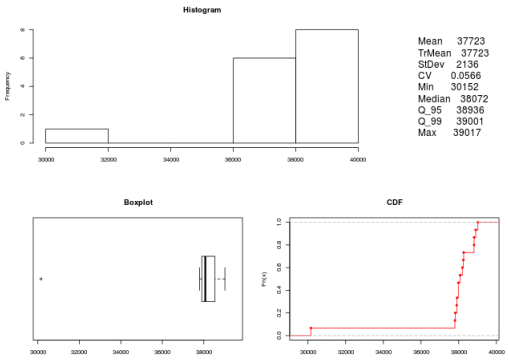
(a) NTFS — Leitura aleatória



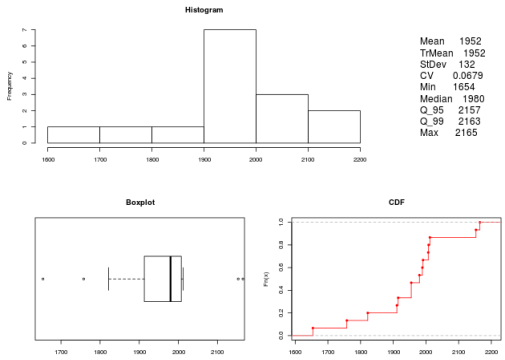
(b) NTFS — Escrita aleatória



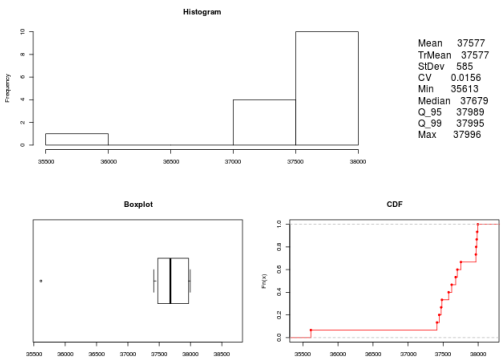
(c) EXT4 — Leitura aleatória



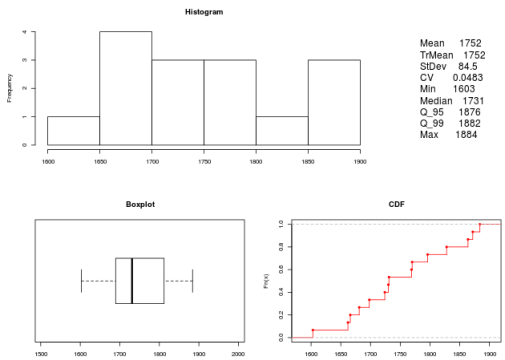
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

Como pode ser observado nas estatísticas das Figuras 11 (c) e (d), o EXT4 apresentou a melhor vazão para a leitura aleatória (min/med/max = 30/38/39 MiB/s) e escrita aleatória (min/med/max = 1,6/1,9/2,1 MiB/s) em um dispositivo *Flashdrive*.

A Tabela 12 apresenta um resumo do desempenho da vazão mediana dos FS em um *Flashdrive*. Na escrita sequencial bem como aleatória o EXT4 apresentou as melhores vazões, sendo elas 1,8 MiB/s e 1,9 MiB/s respectivamente, enquanto que na leitura sequencial o BTRFS apresentou a melhor vazão de 38,4 MiB/s. Por fim o EXT4 se destacou dos FS apresentando um desempenho mediano de 38 MiB/s na leitura aleatória.

Tabela 12 – Vazão mediana dos FS em um *Flashdrive*

IOzone *Benchmark* (*File size*: 1 GiB, *Record length*: 4096 KiB) Vazão (MiB/s)						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	1,57	1,42	0,90	24,78	35,63	1,43
EXT4	**1,87**	**1,98**	1,05	36,61	**38,07**	1,03
BTRFS	1,62	1,73	1,06	**38,44**	37,67	0,97

Fonte: Elaborado pelo autor.

Complementando ainda a análise dos dados, a Tabela 12 nos mostra a razão da escrita aleatória (rnd_W) pela escrita sequencial (W) e, também, a razão da leitura aleatória (rnd_R) pela leitura sequencial (R). Se o resultado dessa operação for maior que 1, o contexto aleatório sobressaiu o sequencial e, do contrário (menor que 1) a vazão sequencial conseguiu atingir um melhor desempenho sobre a aleatória. Podemos ver pelos dados da tabela (12) em questão que o FS EXT4 conseguiu, em ambos os casos (rnd_W e rnd_R), uma vazão melhor neste dispositivo de acesso aleatório. Concluindo, a Tabela 12 nos mostra a razão entre operações aleatórias e sequenciais do FS que apresentou melhor desempenho nesta mídia, que foi de 1,05 em rnd_W/W e de 1,03 em rnd_R/R.

Uma possível explicação pelo do EXT4 ter se destacado no cenário acima é porque este contempla uma estrutura menos complexa que o BTRFS, como argumenta um dos desenvolvedores do *Phoronix Test Suite*[1]

> O BTRFS, com seu comportamento baseado no *copy-on-write*, faz com que ele tenha muitos recursos, mas pelo menos em seu comportamento pronto para uso, em geral, é muito mais lento que o EXT4. Tradução nossa (LARABEL, 2019)

.

Já o fato do EXT4 ter apresentado melhor desempenho que o NTFS, é por questão de que este não é um sistema de arquivos nativo do Linux Kernel, de maneira que a biblioteca que o manipula foi desenvolvida na base de engenharia reversa, o que poderia afetar o desempenho de suas estruturas impactando nas vazões aferidas.

[1] <https://www.phoronix.com/scan.php?page=home>. Acessado em mai/19

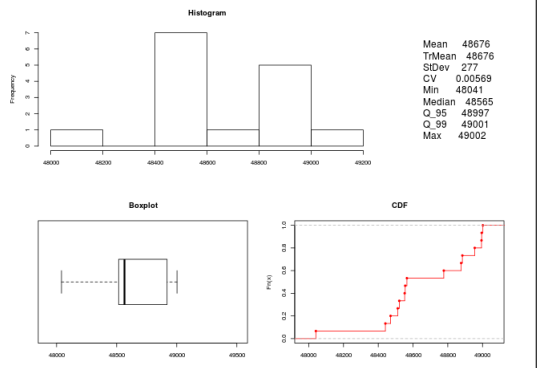
O NTFS é um sistema de arquivos proprietário, e as soluções Unix, Linux, Mac OS e BSD são o resultado de engenharia reversa e implementação deficientes.(WEINTRAUB, 2018)

.

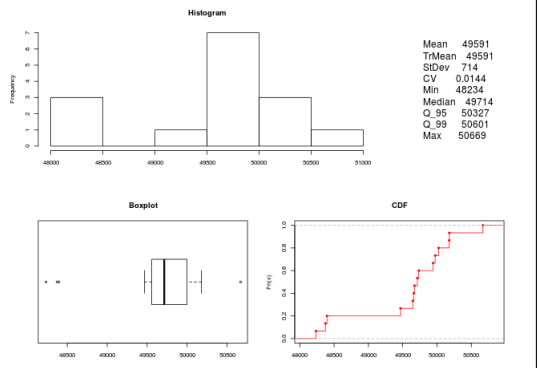
Tabela 13 – Vazão de R/W aleatória dos FS em um HDD

IOzone *Benchmark* (*File size*: 1 GiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

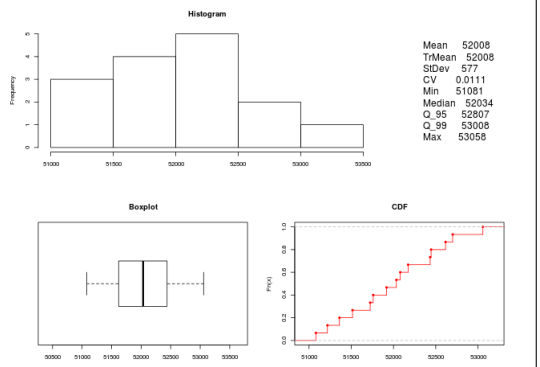
(a) NTFS — Leitura aleatória



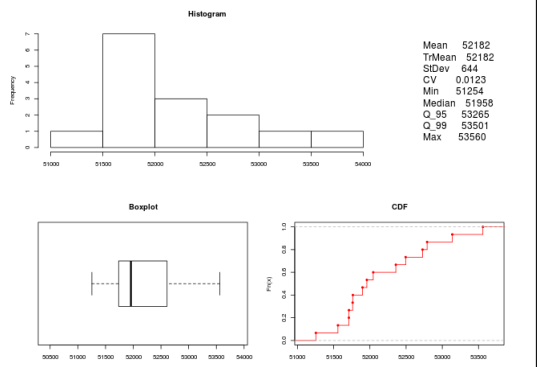
(b) NTFS — Escrita aleatória



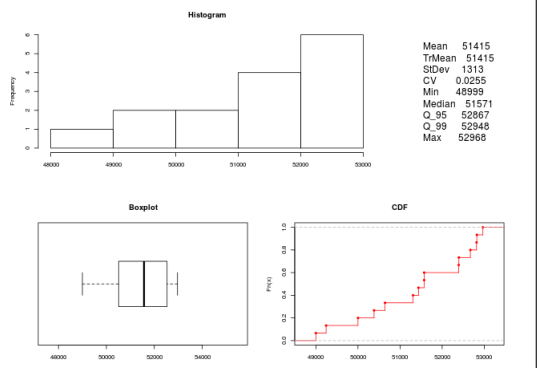
(c) EXT4 — Leitura aleatória



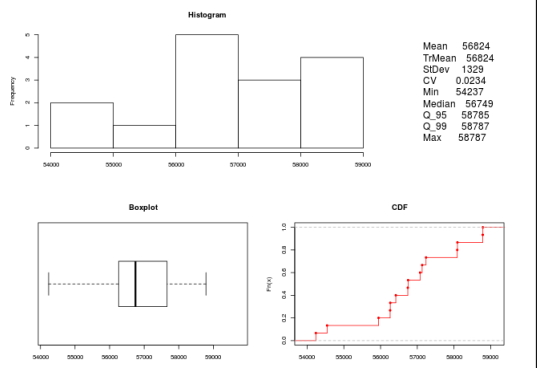
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

Como pode ser observado nas estatísticas das Figuras 13 (e) e (f), o BTRFS apresentou a melhor vazão para a escrita aleatória (min/med/max = 54,2/56,7/58,7

MiB/s) em um dispositivo HDD. Já o EXT4, destaque na mídia anterior (*Pen drive*) obteve um melhor desempenho na leitura aleatória (min/med/max = 51/52/53 MiB/s) em um dispositivo HDD.

A Tabela 14 apresenta um resumo do desempenho da vazão mediana dos FS em um HDD. Na escrita sequencial o BTRFS apresentou a melhor vazão (58,3 MiB/s), enquanto que na leitura sequencial o EXT4 apresentou a melhor vazão (58,4 MiB/s).

Tabela 14 – Vazão mediana dos FS em um HDD

IOzone *Benchmark* (*File size*: 1 GiB, *Record length*: 4096 KiB) Vazão (MiB/s)						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	57,06	49,71	0,87	55,84	48,56	0,86
EXT4	57,52	51,95	0,90	57,30	**52,03**	0,90
BTRFS	**58,39**	**56,74**	0,97	**58,41**	51,57	0,88

Fonte: Elaborado pelo autor.

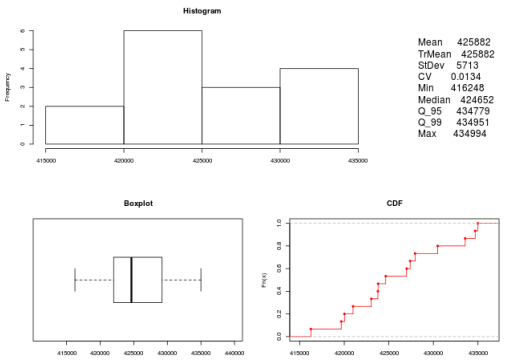
Seguindo a metodologia empregada na explicação dos resultados da Tabela 12, no que diz respeito à razão (rnd_W/W e rnd_R/R), podemos analisar na Tabela 14 que em nenhum dos FS a vazão no contexto aleatório sobressaiu o sequencial, isso se dá pelo fato da mídia em questão ser um HDD e, como este *hardware* opera de forma eletromecânica, o deslocamento necessário para uma escrita/leitura aleatória impacta diretamente em sua vazão. Podemos analisar também na tabela em questão que o FS BTRFS conseguiu o melhor desempenho em relação aos demais, bem como alcançou a razão rnd_W/W de 0,97 e rnd_R/R de 0,88.

Podemos explicar o bom desempenho do BTRFS na mídia em questão pois este utiliza a técnica de COW (ver seção 2.5), mantendo algumas partes que o mesmo utiliza em memória (e.g., superbloco, estrutura de persistência dos dados) podendo fazer sua escrita posteriormente. Esta estratégia beneficia o BTRFS na mídia HDD pois qualquer deslocamento demandado por uma operação poderia fazer com o que seu desempenho do FS caísse drasticamente.

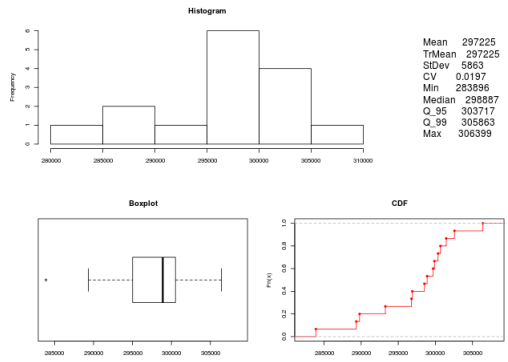
Tabela 15 – Vazão de R/W aleatória dos FS em um SSD

IOzone *Benchmark* (*File size*: 1 GiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

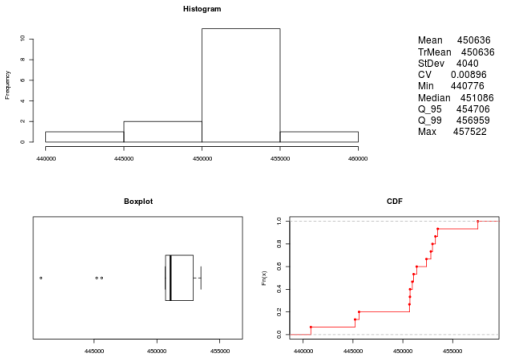
(a) NTFS — Leitura aleatória



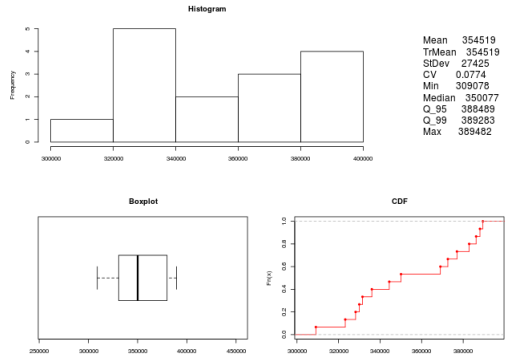
(b) NTFS — Escrita aleatória



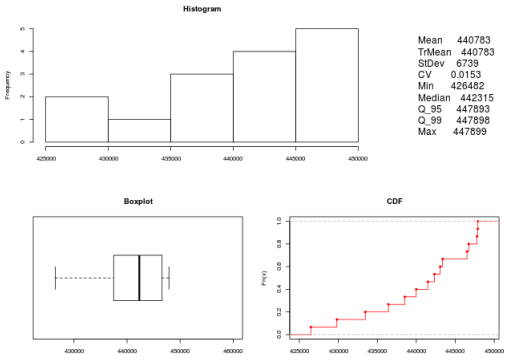
(c) EXT4 — Leitura aleatória



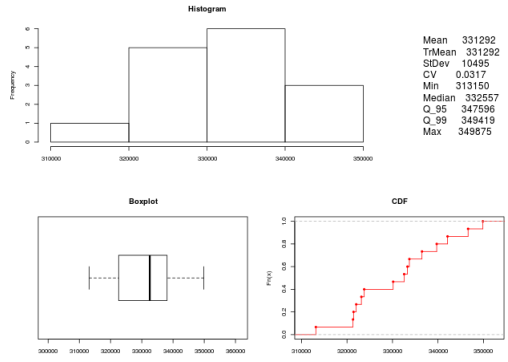
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

Como pode ser observado nas estatísticas das Figuras 15 (c) e (d), o EXT4 apresentou a maior vazão para a leitura aleatória (min/med/max = 440/451/457 MiB/s) e também para a escrita aleatória (min/med/max = 309/350/389 MiB/s) em um dispositivo SSD. A Tabela 16 apresenta um resumo da vazão mediana dos FS em um SSD. Na escrita sequencial o EXT4 apresentou a melhor vazão (330 MiB/s), enquanto que na leitura sequencial o BTRFS apresentou a melhor vazão (503 MiB/s).

Tabela 16 – Vazão mediana dos FS em um SSD

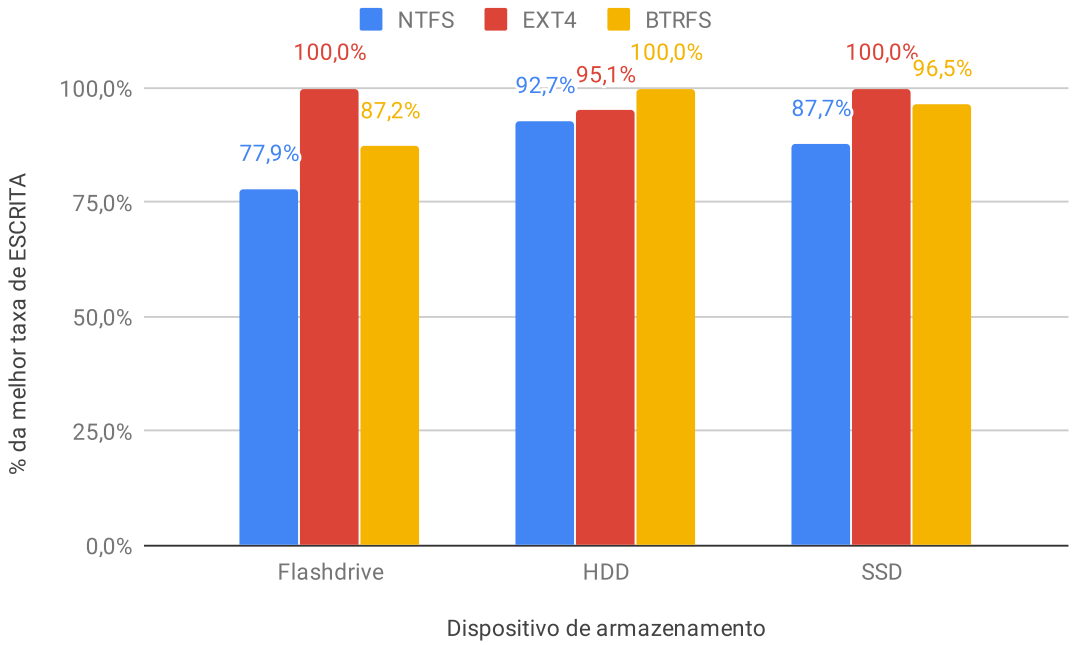
IOzone *Benchmark* (*File size*: 1 GiB, *Record length*: 4096 KiB) Vazão (MiB/s)						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	297,98	298,88	1,00	463,46	424,65	0,91
EXT4	**330,30**	**350,07**	1,05	486,87	**451,08**	0,92
BTRFS	323,71	332,55	1,02	**503,09**	442,31	0,90

Fonte: Elaborado pelo autor.

Considerando o cenário acima (SSD), destacamos o desempenho do EXT4 com a razão rnd_W/W de 1,05 e a rnd_R/R de 0,92 em um dispositivo com acesso aleatório, pela mesma justificativa apresentada para a Tabela 12.

Podemos ver pelos testes realizados, que o sistema de arquivos EXT4 (*Fourth Extended File System*) conseguiu obter uma maior vazão em dispositivos de acesso aleatório (*FlashDrive* e o SSD), de maneira que o EXT4 foi escolhido para usarmos como o FS subjacente com o menor *overhead* para o teste de desempenho do FS desenvolvido neste trabalho.

Figura 24 – Desempenho do FS: escrita por tipo de dispositivo

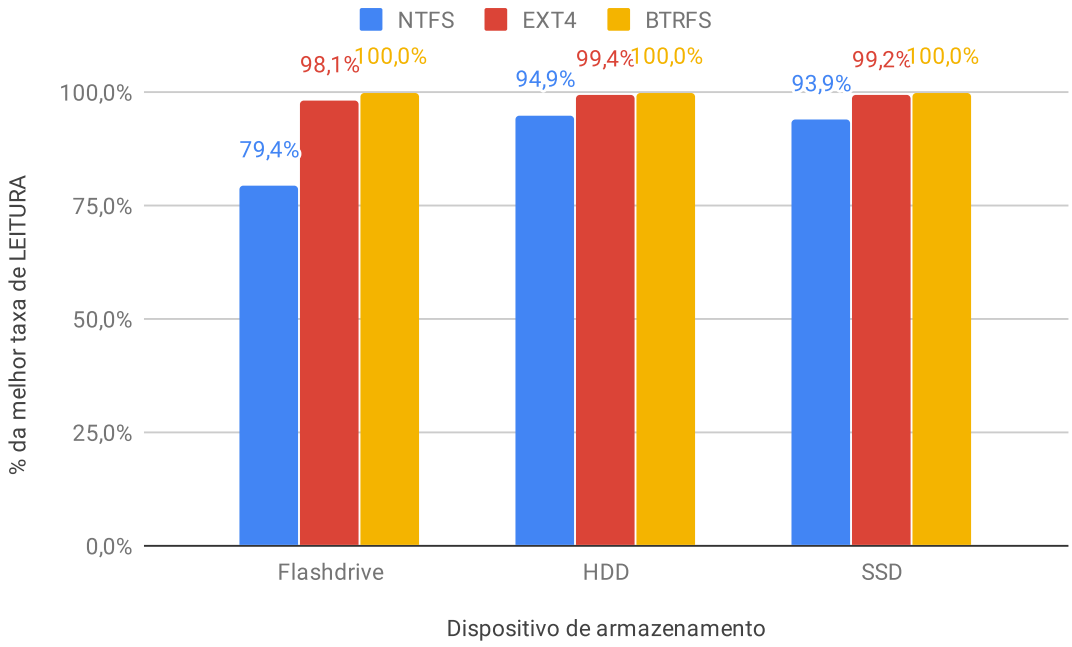


Fonte: Elaborado pelo autor.

Considerando o gráfico da Figura 24, notamos que o EXT4 obteve o melhor desempenho para **operações de escrita** em ambos os dispositivos *Flash drive* e SSD,

por serem mídias de acesso aleatório. Já o BTRFS apresentou um desempenho melhor apenas no dispositivo HDD, no qual apresentou uma taxa de escrita 4,9% melhor que o EXT4. Novamente, o NTFS apresentou o **pior desempenho em todos os dispositivos** de armazenamento testados, com desempenho de escrita 12,3–22,1% inferior que o EXT nos dispositivos *Flash drive* e SSD.

Figura 25 – Desempenho do FS: leitura por tipo de dispositivo



Fonte: Elaborado pelo autor.

Por outro lado, podemos analisar pelo gráfico da Figura 25 que o BTRFS obteve o melhor desempenho para **operações de leitura** nos dispositivos (*Flash drive*, HDD e SSD). Em segundo lugar observa-se o EXT4, com um desempenho de leitura apenas de 0,8–1,9% inferior ao BTRFS. Por fim ressaltamos o NTFS, por ter obtido o **pior desempenho em todos os dispositivos** de armazenamento testados. No dispositivo *Flashdrive*, o NTFS apresentou um desempenho 20,6% inferior em relação à melhor taxa de leitura obtida (BTRFS).

5.2.2 Desempenho do FS desenvolvido

Este presente tópico tem como finalidade apresentar e discutir os resultados dos experimentos conduzidos com o sistema de arquivo desenvolvido.

Com base nos testes realizados na subseção anterior (5.2), definimos o EXT4 como sendo o FS mais apropriado para suportar a realização dos testes de desempenho do FS desenvolvido. Novamente ressalto que o uso dessa abordagem é devido ao fato do prazo

da realização de um TCC, pois a o ideal seria colocar o nosso FS em espaço do *Kernel*. Entretanto, esta etapa não é nenhum pouco trivial e demandaria uma dedicação de tempo inviável. Voltando ao assunto em questão, ao executarmos o nosso FS sobreposto ao FS que apresentou melhor desempenho, será possível observar o comportamento do mesmo em espaço de usuário, nas mídias (*Pen Drive*, HDD e SSD), sendo que o FS subjacente realizará as operações em espaço de *Kernel*.

Como o FS desenvolvido foi testado sobreposto ao EXT4 (FS subjacente), toda operação de leitura ou escrita será realizada por ambos, inicialmente no disco virtual e posteriormente no disco físico. Observe que interrupções e escalonamentos no sistema operacional podem afetar as medições. Por isso, foram realizadas 15 repetições para cada operação avaliada (escrita sequencial, escrita aleatória, leitura sequencial e leitura aleatória) do experimento. Tal experimento foi realizado manualmente e separadamente sem o uso de qualquer tarefa do usuário no computador utilizado nos testes, a fim de minimizar o impacto de tais interferências externas.

O conteúdo do arquivo a ser gravado no FS é extraído diretamente da memória principal (RAM) e gravado em um arquivo binário (disco virtual) que simula o disco físico nas mídias descritas (*Pen Drive*, HDD e SSD). Vale ressaltar que a integridade dos arquivos no FS desenvolvido é verificada, conforme descrito na seção 3.2.4. Toda operação (de leitura ou escrita) será inicialmente manipulada pelas estruturas do FS desenvolvido porém, observe que o arquivo binário (disco virtual) será efetivamente operado pelo FS subjacente, no caso o EXT4.

Antes de apresentar os resultados aferidos, vale comparar as estruturas de ambos os FS (o desenvolvido, bem como o subjacente EXT4). A Tabela 17 apresenta as estruturas dos FS lado a lado.

Tabela 17 – Estrutura dos FS

Estruturas do FS desenvolvido vs Estrutura do EXT4		
Estrutura	**Meu FS**	**EXT4**
Superbloco	Estrutura heterogenea de dados	Estrutura heterogenea de dados
Gerenciamento do espaço livre	Bitmap (0 vazio, 1 ocupado)	Bitmap (0 vazio, 1 ocupado)
Diretórios	Estrutura tipo árvore	*Hash table* e estrutura tipo árvore
Inode	Estrutura tipo árvore	O EXT4 salva na entrada do diretório uma cópia dos *inode* em questão. Matriz Linear (Vetor)

Fonte: Elaborado pelo autor.

Vale atentar também que para a medição do nosso FS o caminho padrão utilizado para todos os arquivos foi na raiz do diretório ("/"), este que também é o caminho utilizado

nas mídias que este trabalho aborda. Esta tática foi pensada para diminuir o uso da estrutura, uma vez que a operação na raiz a mesma que em outra pasta.

5.2.2.1 Cenário 1 - Arquivos de tamanho pequeno (10 MiB)

O presente cenário tem como objetivo mostrar como o FS desenvolvido se comporta ao manipular dados relativamente pequenos, cujos *bytes* são gerados aleatoriamente para formar um arquivo de 10 MiB a ser utilizado no teste. Este cenário busca simular o uso do FS para armazenamento de músicas, documentos de texto, bibliotecas de programas ou qualquer outro uso que comporte arquivos de tamanho pequeno (até 10 MiB).

A Tabela 18, demonstra a mediana da vazão aferida no cenário de arquivos de tamanho pequeno. O FS desenvolvido se beneficiou de escrita aleatória em dispositivo de armazenamento **SSD** e, inversamente, se beneficiou menos de leitura sequencial em dispositivo HDD. Tal fato reflete as decisões de projeto do FS experimental uma vez que foi elaborado pensando em dispositivos de armazenamento de acesso aleatório (em especial, SSD).

Tabela 18 – Desempenho do FS desenvolvido - Arquivos de 10 MiB

Desempenho do FS desenvolvido para arquivos de 10 MiB Vazão (MiB/s) — Mediana						
Mídia	Write	Random Write	Razão rnd_W / seq_W	Read	Random Read	Razão rnd_R / seq_R
FlashDrive	3,98	3,31	*0,83*	26,11	25,68	*0,98*
HDD	37,51	36,45	0,97	47,13	59,99	**1,27**
SSD	311,56	344,81	**1,10**	381,14	391,56	1,02

Fonte: Elaborado pelo autor.

Vale ressaltar que, como estamos rodando o nosso FS sobreposto ao EXT4, seria impossível obter uma vazão acima daquela que foi medida no próprio EXT4 (que roda em nível *Kernel*). Portanto, discorreremos sobre as vazões medidas em nosso FS em nível de perda de desempenho, ou seja, o quão menos eficiente foram nossas estruturas em relação ao EXT4 “puro”.

A Tabela 19 demonstra em porcentagem a perda de desempenho em relação ao EXT4 nas diferentes mídias de armazenamento. A perda de desempenho mais acentuada foi em operações de escrita aleatória em dispositivo de armazenamento *FlashDrive* (54,01%), enquanto que a menor perda de desempenho foi para operações de leitura sequencial em dispositivos HDD e **SSD**. Considerando o pequeno tamanho dos arquivos de teste, talvez isto poderia ser explicado pela própria “bufferização” realizada pelos dispositivos de armazenamento, visando adiantar-se à possíveis operações de leitura subsequente.

Tabela 19 – Perda de desempenho do FS desenvolvido sobre o EXT4 - Arquivos de 10 MiB

Perda de desempenho (%) do FS desenvolvido sobre o EXT4. Arquivo de 10 MiB.				
Mídia	Write	Random Write	Read	Random Read
FlashDrive	39,32 %	**54,01 %**	14,25 %	19,91 %
HDD	9,23 %	14,26 %	8,73 %	11,58 %
SSD	11,00 %	15,91 %	8,76%	12,69 %

Fonte: Elaborado pelo autor.

5.2.2.2 Cenário 2 - Arquivos de Tamanho Intermediário (100 MiB)

Neste cenário é utilizada a mesma abordagem, porém diferentemente do experimento anterior, em um arquivo de tamanho intermediário para simbolizar casos de uso do nosso FS com, por exemplo, arquivos multimídia (fotos, vídeos, livros, dentre outros). A Tabela 20, mostra a vazão aferida neste cenário. Neste caso, o melhor aproveitamento do acesso aleatório foi obtido em operações de escrita aleatória em dispositivo **SSD** (resultado consistente com o cenário anterior). No caso de operações de leitura, o FS desenvolvido se beneficiou mais do acesso aleatório em dispositivo ***FlashDrive***.

Tabela 20 – Desempenho do FS desenvolvido - Arquivos de 100 MiB

Desempenho do FS desenvolvido para arquivos de 100 MiB Vazão (MiB/s) — Mediana						
Mídia	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
FlashDrive	1,83	1,74	0,94	28,06	29,34	**1,04**
HDD	51,12	42,45	*0,83*	51,25	42,88	*0,83*
SSD	322,30	315,59	**0,97**	433,74	415,58	0,95

Fonte: Elaborado pelo autor.

Quanto à perda relativa de desempenho do FS desenvolvido em relação ao EXT4, conforme pode ser observado na Tabela 21 novamente o FS demonstrou pior desempenho em dispositivo *FlashDrive*. Por outro lado, a menor perda de desempenho observada foi em dispositivo **SSD** para operações de escrita aleatória e, no caso de leitura sequencial, no dispositivo HDD (como seria de se esperar dado o mecanismo de operação desta mídia).

Tabela 21 – Perda de desempenho do FS desenvolvido sobre o EXT4 - Arquivos de 100 MiB

Perda de desempenho (%) do FS desenvolvido sobre o EXT4. Arquivo de 100 MiB.				
Mídia	Write	Random Write	Read	Random Read
FlashDrive	13,88 %	17,11 %	12,5 %	**21,63** %
HDD	7,19 %	15,03 %	8,73 %	11,97 %
SSD	13,35 %	15,09 %	8,83 %	7,84 %

Fonte: Elaborado pelo autor.

5.2.2.3 Cenário 3 - Arquivos Grandes (1000 MiB)

Por fim, seguindo a metodologia dos testes, este cenário tem como objetivo demonstrar como o nosso FS se comporta ao manipular arquivos grandes, estes podendo ser filmes, artefatos de jogos, arquivos empresariais (*log* e *csv*), bem como arquivos de *backup*.

Tabela 22 – Desempenho do FS desenvolvido - Arquivos de 1000 MiB

Desempenho do FS desenvolvido para arquivos de 1000 MiB Vazão (MiB/s) — Mediana						
Mídia	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
FlashDrive	1,25	1,20	0,95	29,44	28,74	**0,97**
HDD	52,22	46,56	*0,89*	52,17	47,90	*0,91*
SSD	319,74	317,74	**0,99**	438,74	415,57	0,94

Fonte: Elaborado pelo autor.

Conforme pode ser observado na Tabela 22, o sistema de arquivo (experimental) desenvolvido neste trabalho obteve desempenho consistente se comparadas operações de leitura/escrita sequenciais e aleatórias. Como era de se esperar, para o dispositivo de armazenamento HDD o "benefício" em operações de leitura aleatória e de escrita aleatória em relação à correspondente operação sequencial foi menor, dado o mecanismo de operação desse tipo de mídia. Já comparando-se os dispositivos de armazenamento com acesso aleatório, o FS desenvolvido obteve melhor desempenho em operações de escrita no **SSD** do que em *FlashDrive*, bem como em operações de leitura em ***FlashDrive*** do que no SSD.

Tabela 23 – Perda de desempenho do FS desenvolvido sobre o EXT4 - Arquivos de 1000 MiB

Perda de desempenho (%) do FS desenvolvido sobre o EXT4. Arquivo de 1000 MiB.				
Mídia	Write	Random Write	Read	Random Read
FlashDrive	32,77 %	**39,2 %**	19,58 %	24,51 %
HDD	9,23 %	10,38 %	8,96 %	7,95 %
SSD	3,2 %	9,24 %	9,89 %	*7,88* %

Fonte: Elaborado pelo autor.

Por fim, quanto à perda de desempenho em relação ao EXT4, conforme pode ser observado na Tabela 23 o FS consistentemente demonstrou pior desempenho em dispositivos do tipo *FlashDrive*. Por outro lado, as menores perdas de desempenho observadas foram em dispositivo **SSD** tanto para operações de leitura aleatória quanto para operações de escrita sequencial, resultado parecido com o do cenário anterior.

Pode-se observar, portanto, que o FS experimental desenvolvido demonstrou se beneficiar mais de operações de acesso aleatório em dispositivo **SSD**, bem como apresentou uma considerável queda de desempenho no caso de dispositivo *FlashDrive* em relação ao sistema de arquivo EXT4. Tal fato reflete as decisões de projeto do FS experimental, uma vez que foi elaborado pensando em dispositivos de armazenamento do tipo SSD.

6 CONCLUSÕES E TRABALHOS FUTUROS

6.1 Conclusões

O presente trabalho teve como objetivo o desenvolvimento de um sistema de arquivos para dispositivos de armazenamento com acesso aleatório, baseado na ideia de paginas para a exploração das mídias que contam com o acesso aleatório. Para validação das estruturas verificamos a integridade de arquivos reais (pdf, mp3, cvs), que foram inseridos e recuperados com sucesso, garantindo o funcionamento operacional do nosso FS.

Para a realização do projeto experimental, foram selecionados três tipos de cenários, cada um deles contendo diferentes mídias de armazenamento: um *Flash Drive*, um SSD de acesso aleatório e um HDD de acesso sequencial, este último sendo necessário pois ainda tem um grande uso no contexto atual. O primeiro cenário consistiu em mostrar o desempenho do FS em arquivos de tamanho pequenos; o segundo em arquivos de tamanho intermediários; e por fim o uso do nosso FS em arquivos grandes. Após a execução dos experimentos foi possível perceber como o nosso FS atua ao tratar diferentes tamanhos de arquivos em sua estrutura.

Um outro experimento realizado para explorar o conhecimento científico foi a medição de desempenhos de alguns FS de uso frequente (NTFS, EXT4 e BTRFS), com finalidade de aferir em qual FS se obtêm a maior vazão tanto na escrita sequencial/aleatória e leitura sequencial/aleatória, abrindo uma outra oportunidade de estudo.

Isto posto, conclui-se que o uso da alocação de blocos bem como as operações do FS abordados na metodologia deste trabalho são boas abordagens para serem seguidas e desenvolvidas gerando uma nova perspectiva para sugerir o desenvolvimento de sistemas de arquivos para dispositivos de armazenamento que tem o seu foco em mídias com o acesso aleatório. Entretanto, como o uso do FS desenvolvido sobrepõem um FS em questão, os dados coletados são dados inconclusivos para grandes generalizações, necessitando a implementação do mesmo em espaço do *Kernel* para alcançar resultados mais satisfatórios em desempenho. Estas e outras futuras explorações serão descritas na seção posterior, deixando algumas sugestões para o incremento deste projeto.

6.2 Trabalhos Futuros

Como o desenvolvimento deste trabalho atinge o nível de maturidade tecnológica TRL-5: Teste laboratorial da integração dos componentes, em ambiente relevante (ex-

perimento em cenário virtual/sintético) ver 2.8, ainda há trabalhos a serem feitos para que atinja o topo desta metodologia (TRL-9 Produto completo, pronto para implantação/comercialização em larga escala e disponível aos usuários finais).

Como foi abordado na seção 6.1, este trabalho abre duas novas oportunidades para o estudo e desenvolvimento de novos trabalhos, uma para os que querem seguir com a implementação do FS em questão, e outra para os que querem seguir o ramo de análise de sistemas de arquivos em geral.

Para os que querem usar a abordagem deste trabalho para a continuação do desenvolvimento do FS, reforço que os resultados das vazões medidas do FS desenvolvido nas presentes mídias ainda são inconclusivos, pois existe uma estrutura superior que de fato manipula os dados e, portanto, não refletem o real desempenho do sistema de arquivos proposto. Para trabalhos futuros encontram-se as seguintes sugestões a seguir:

- Implementar a metologia seguida neste trabalho em ambiente operacional do Linux, que exige a recompilação do SO juntamente com a integração da proposta. Ao realizar esta operação os valores medidos serão dados como reais, pois não existirá nenhuma limitação ou outra estrutura para influenciar nas medições, atingindo ao final do projeto mais dois níveis do *ranking* da maturidade tecnológica, alcançando a TRL-7 (ver Tabela 2).

- Abordar uma estrutura ou mecanismo de segurança (em nível de consistência) dos dados para maximizar a vida útil das mídias que utilizam memória *flash* juntamente com nosso FS, pois por mais que o nosso FS desenvolvido garanta a persistência dos dados (reescrevendo o superbloco a cada operação), o uso desta tática é dada como precária, pois qualquer corrompimento do superbloco ao fazer uma operação pode fazer com que os dados que estão sendo gravados se percam, gerando inconsistência. O uso desta tática em questão não aborda a vida útil da mídia.

- Também é necessário implementar o restantes das funções do VFS, para tornar o FS desenvolvido mais utilizável e escalável, pois as funções implementadas neste trabalho foram somente os objetos primários de um FS. Detalhes dessas operações podem ser vistos no capítulo 4. Ao atingir este requisito o FS desenvolvido estará bem próximo da produção completa atingindo o nível de maturidade TRL-8 (ver Tabela 2), sendo necessário apenas validar as operações implementadas, garantindo a consistência de todo o funcionamento do FS, para que então alcance a TRL-9.

Para os que querem seguir a outra área de conhecimento que este projeto aborda, que são as medições e comparações dos FS (NTFS, EXT4 e BTRFS) já consolidados, deixo aqui a seguir as seguintes sugestões:

- Ao medir o desempenho dos FS, também medir o uso de processamento e gasto de memória em suas operações tanto de escrita sequencial, escrita aleatória, leitura sequencial e leitura aleatória.

- Incluir outros FS, que por agora encontram desenvolvidos mas não totalmente otimizados e consolidados, como por exemplo o F2FS, XFS, entre outros. Ao incluir novos FS também testá-los fazendo o uso de *RAID* como mídia física.

- Fazer o uso do IOzone por completo (ver seção 3.1.2.1), pois este *software* oferece ainda mais recursos que, para este trabalho, não foram relevantes mas que para comparações entre FS podem ser úteis. Se optarem por o uso de outros *software*, informo que ainda existe na literatura o Bonnie++[1], não abordado neste projeto.

[1] <https://www.coker.com.au/bonnie++/readme.html>. Acessado em mai/19

REFERÊNCIAS

ANDRADE Érika. *O PDCA COMO FERRAMENTA DE GESTÃO DA ROTINA*. 2015. Disponível em: <http://www.inovarse.org/sites/default/files/T_15_017M_7.pdf>. Acesso em: 04 out. 2018. Citado na página 39.

ANDREY, M. *The virtual File System (VFS)*. 2005. Disponível em: <https://www.cos.ufrj.br/~vitor/aulas/COS773/alunos/VFS.pdf>. Acesso em: 11 mai. 2018. Citado na página 24.

BORISLAV DJORDJEVIC; VALENTINA, T. *Ext4 file system in Linux Environment: Features and Performance Analysis*. 2012. 9 p. Citado 3 vezes nas páginas 45, 46 e 47.

CARVALHO, R. P. *Sistema de arquivos paralelos: alternativas para a redução do gargalo no acesso ao sistema de arquivos*. 131 f. Monografia (Mestrado) — Universidade de São Paulo, São Paulo, 2005. Citado 2 vezes nas páginas 20 e 21.

COUTINHO, T. *O que é o ciclo PDCA?* 2017. Disponível em: <https://www.voitto.com.br/blog/artigo/o-que-e-o-ciclo-pdca>. Acesso em: 14 set. 2018. Citado na página 39.

CRUZ, O. D. Recuperação de dados em sistema de arquivos ext4. Brasília, f. 21, 2015. Citado 2 vezes nas páginas 20 e 29.

EMBRAPII. *MANUAL DE OPERAÇÃO DAS UNIDADES EMBRAPII*. 2016. Disponível em: <http://embrapii.org.br/wp-content/uploads/2014/10/manual_embrapii_unidades_versao_4-0_final_revisado.pdf>. Acesso em: 20 mai. 2019. Citado na página 34.

EMPREENDEDORA, A. *MVP – MÍNIMO PRODUTO VIÁVEL*. 2017. Disponível em: <https://aliancaempreendedora.org.br/tamojunto/wp-content/uploads/2017/04/MVP.pdf>. Acesso em: 20 mai. 2019. Citado na página 33.

ENDEAVOR, B. *5 dúvidas básicas sobre fazer um protótipo para seu negócio*. 2015. Disponível em: <https://endeavor.org.br/estrategia-e-gestao/prototipo/>. Acesso em: 20 mai. 2019. Citado na página 33.

FLACH, T. K. *Desenvolvimento de um sistema de arquivos instalável para Linux*. 72 f. Monografia (Graduação) — Universidade Regional de Blumenau, Blumenau, 2009. Citado 5 vezes nas páginas 21, 22, 23, 25 e 26.

FUSE. *Filesystem in userspace*. 2009. Disponível em: <https://github.com/libfuse/libfuse>. Acesso em: 04 set. 2018. Citado na página 26.

HUDSON, A. *NTFS: A File System with Integrity and Complexity*. 2010. Disponível em: <https://www.hardware.com.br/artigos/ntfs/>. Acesso em: 05 out. 2018. Citado na página 31.

JAN-NICLAS HILGERT; MARTIN, L. S. Y. b. Forensic analysis of multiple device btrfs configurations using the sleuth kit. Houston, TX, USA, f. 9, 2018. Citado 2 vezes nas páginas 48 e 49.

JIAN, J. A log-structured file system for hybrid volatile/non-volatile main memories. San Diego, f. 17, 2016. Citado na página 32.

JONES, T. *Anatomia do sistema de arquivos do Linux.* 2007. Disponível em: <https://www.ibm.com/developerworks/br/library/l-linux-filesystem/index.html>. Acesso em: 24 set. 2018. Citado na página 23.

JONES, T. *Anatomy of ext4.* 2009. Disponível em: <https://www.ibm.com/developerworks/library/l-anatomy-ext4/l-anatomy-ext4-pdf.pdf>. Acesso em: 27 set. 2018. Citado na página 45.

KASAMPALIS, S. Copy on write based file systems performance analysis and implementation. Technical University of Denmark, f. 94, 2010. Citado na página 30.

LARABEL, M. *Linux 5.0 File-System Benchmarks: Btrfs vs. EXT4 vs. F2FS vs. XFS.* 2019. Disponível em: <https://www.phoronix.com/scan.php?page=article&item=linux-50-filesystems&num=4>. Acesso em: 15 mai. 2019. Citado na página 61.

LIMA, L. *Sistemas de arquivos.* 2012. Disponível em: <http://www.ppgia.pucpr.br/~laplima/ensino/so/materia/04_arquivos.html>. Acesso em: 16 mai. 2018. Citado na página 20.

LIRA, L. de. Revisão bibliométrica sobre a produção científica em aprendizagem gerencial. Rio de Janeiro, f. 21, 2010. Citado na página 38.

LOPES, N. L. *Sistema de Ficheiros Transaccional sobre FUSE.* 103 f. Monografia (Mestrado) — Universidade Nova de Lisboa, Lisboa, 2009. Citado na página 27.

LOVE, R. *Desenvolvimento do kernel do Linux Tradução Eveline Vieira Machado Edição, 1ª edição.* [S.l.]: Ciência Moderna, 2004. 355 p. Citado 2 vezes nas páginas 16 e 22.

MACHADO F.; MAIA, L. *Arquitetura de sistemas operacionais, 4ª edição.* [S.l.]: LTC, 2007. 324 p. Citado na página 16.

MASON, T. *Why does Windows still use NTFS?* 2018. Disponível em: <https://www.quora.com/Why-does-Windows-still-use-NTFS-Why-not-ext4-the-file-system-for-Linux-since-it-actively-prevents
Acesso em: 19 mai. 2019. Citado na página 17.

MICROSOFT. *How NTFS Works.* 2009. Disponível em: <https://docs.microsoft.com/en-us/previous-versions/windows/>. Acesso em: 05 out. 2018. Citado 3 vezes nas páginas 31, 43 e 44.

MOREIRA, E. *Saiba por que a Prova de Conceito (POC) é importante.* 2017. Disponível em: <http://introduceti.com.br/blog/saiba-por-que-a-prova-de-conceito-poc-e-importante/>. Acesso em: 20 mai. 2019. Citado na página 33.

POSSAMAI, C. S. *Protótipo de gerenciador de arquivos para ambiente distribuído.* 107 f. — Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 1999. Citado na página 16.

PRACIANO, E. *Introdução ao sistema de arquivos BTRFS.* 2016. Disponível em: <https://elias.praciano.com/2016/02/introducao-ao-sistema-de-arquivos-btrfs>. Acesso em: 12 out. 2018. Citado na página 30.

RENATO, S. *QUANTO TEMPO DE VIDA TEM O SEU SSD?*. 2017. Disponível em: <https://olhardigital.com.br/noticia/quanto-tempo-de-vida-tem-o-seu-ssd-descubra-agora/65923>. Acesso em: 10 mai. 2018. Citado na página 17.

ROBERTO, B. Introduction to flash memory. f. 14, 2003. Citado na página 32.

SILBERSCHATZ, A. *fundamentos de sistemas operacionais,8ª edição*. [S.l.]: LTC, 2009. 536 p. Citado 4 vezes nas páginas 20, 21, 22 e 39.

TANENBAUM ANDREW S.; WOODHULL, A. S. *Sistemas operacionais: projeto e implementação,3ª edição*. [S.l.]: Bookman, 2008. 985 p. Citado na página 21.

WEINTRAUB, J. *Why does Windows still use NTFS?* 2018. Disponível em: <https://www.quora.com/Why-does-Windows-still-use-NTFS-Why-not-ext4-the-file-system-for-Linux-since-it-actively-prevents
Acesso em: 19 mai. 2019. Citado na página 62.

APÊNDICE A – TABELAS(GRAFICO DOS DESEMPENHOS DOS FS)

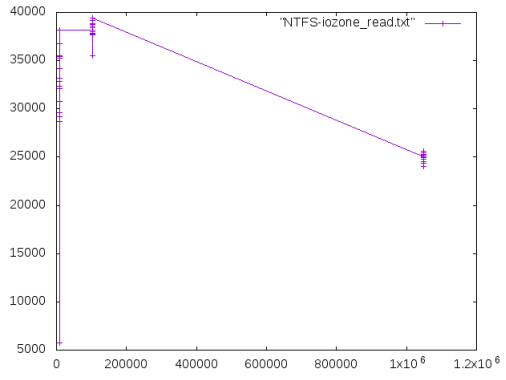
As Tabelas 24, 25, 26 que correspondem ao NTFS, Tabelas 27, 28, 29 correspondem ao EXT4 e, por fim, as Tabelas 30, 31 e 32 são referentes ao BTRFS. Tais tabelas tem como finalidade apresentar as medições realizadas pelo do IOzone dos FS nas respectivas mídias (*Pen Drive*, HDD e SSD), usando um shell script gnuplot (Ver seção 3 materiais, subseção 3.1.2.4). O eixo horizontal se dá pelo tamanho do arquivo e na vertical a vazão do sistema de arquivos.

A.0.0.1 Desempenho dos FS com arquivo de 10, 100 e 1000 MiB (v1)

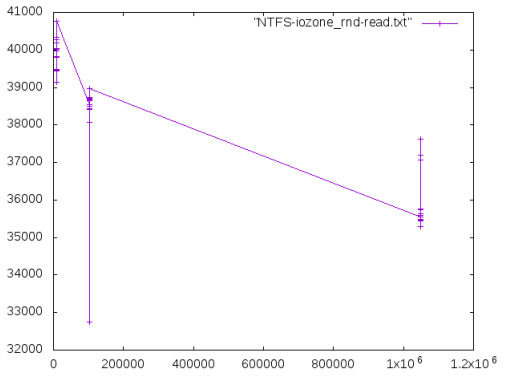
Tabela 24 – Desempenho do NTFS em um PenDrive

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do NTFS (KiByte/s)

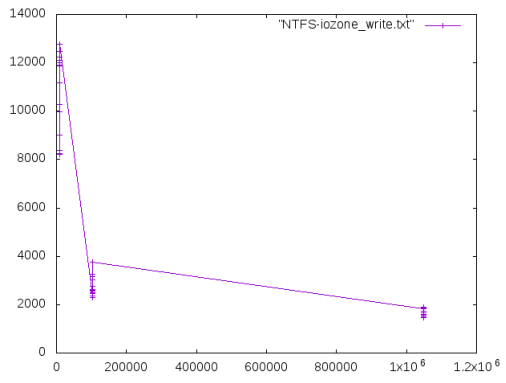
(a) Leitura sequencial



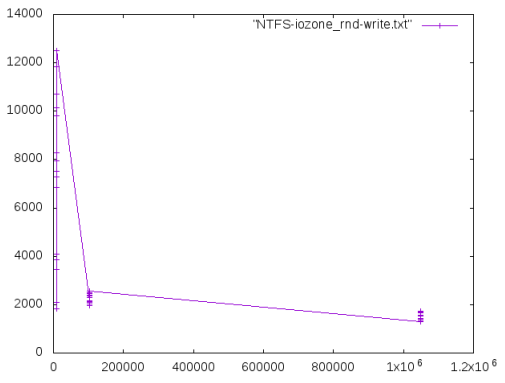
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória

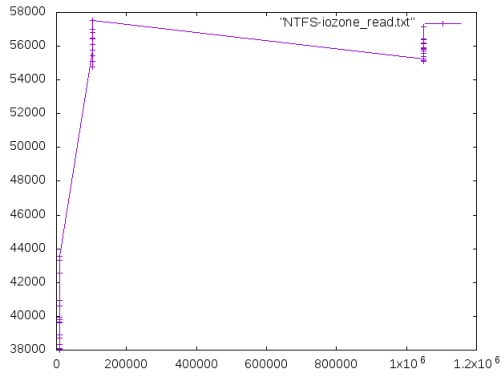


Fonte: Elaborado pelo autor.

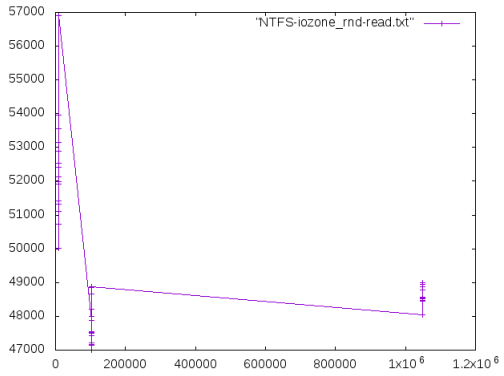
Tabela 25 – Desempenho do NTFS em um HDD

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do NTFS (KiByte/s)

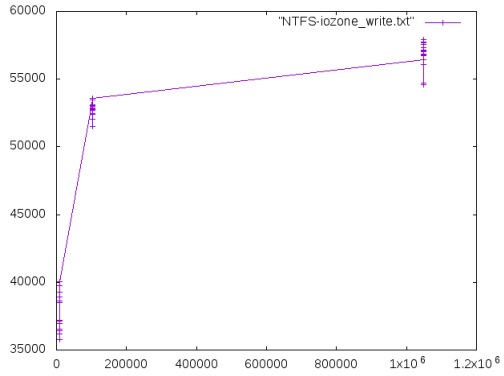
(a) Leitura sequencial



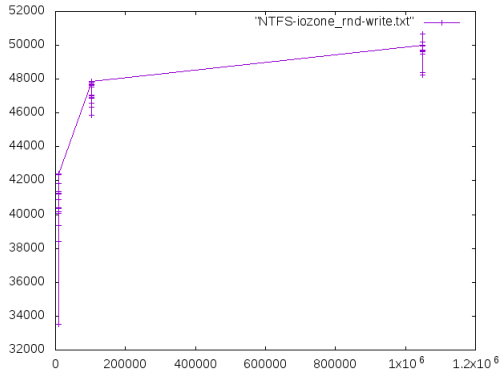
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória

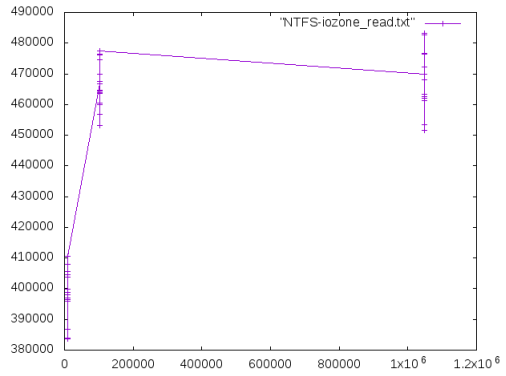


Fonte: Elaborado pelo autor.

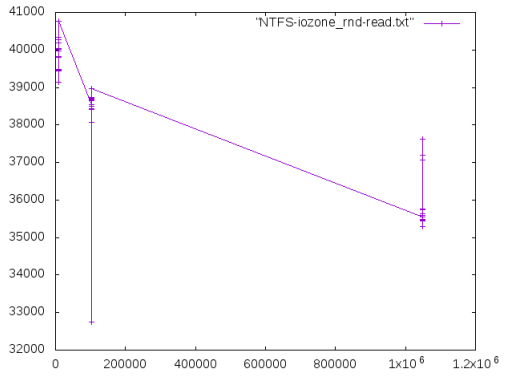
Tabela 26 – Desempenho do NTFS em um SSD

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do NTFS (KiByte/s)

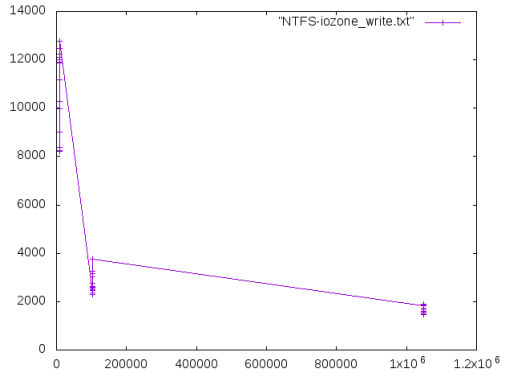
(a) Leitura sequencial



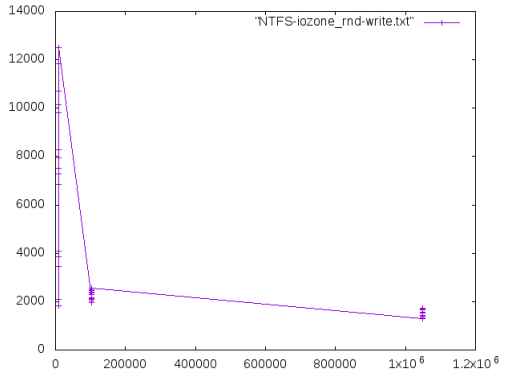
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória

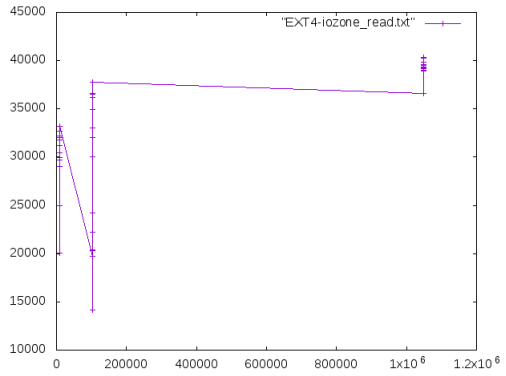


Fonte: Elaborado pelo autor.

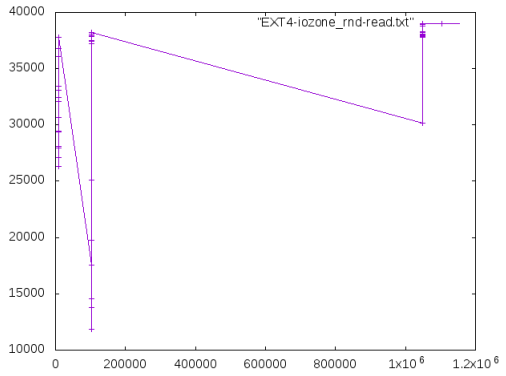
Tabela 27 – Desempenho do EXT4 em um *PenDrive*

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do EXT4 (KiByte/s)

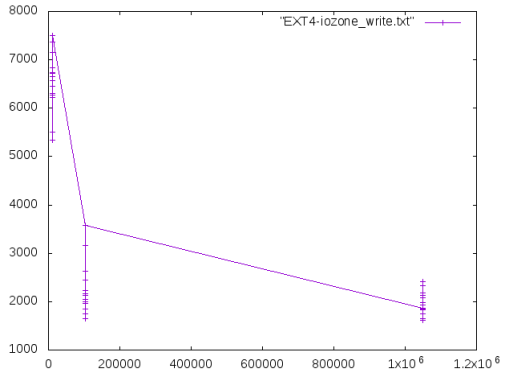
(a) Leitura sequencial



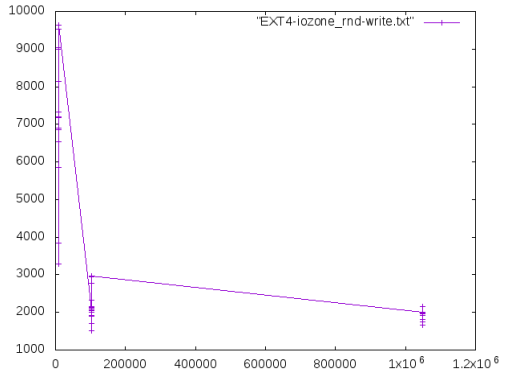
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória

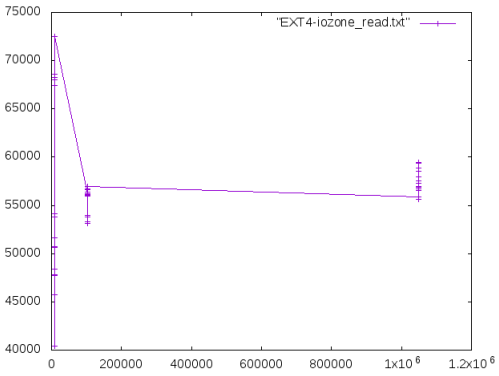


Fonte: Elaborado pelo autor.

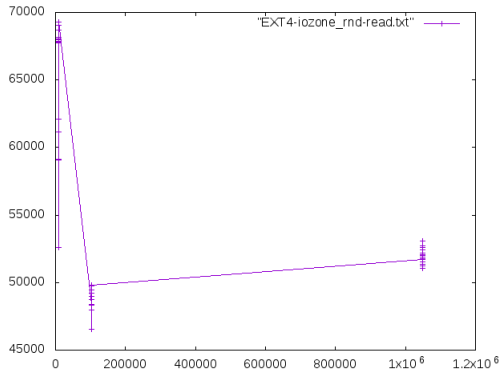
Tabela 28 – Desempenho do EXT4 em um HDD

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do EXT4 (KiByte/s)

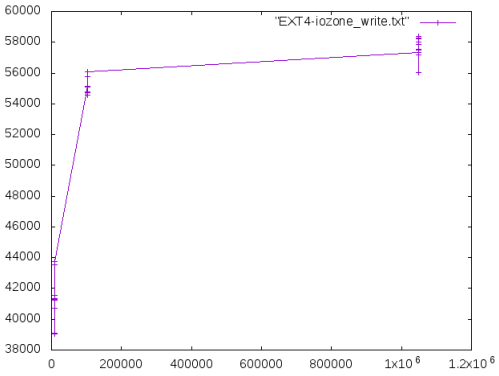
(a) Leitura sequencial



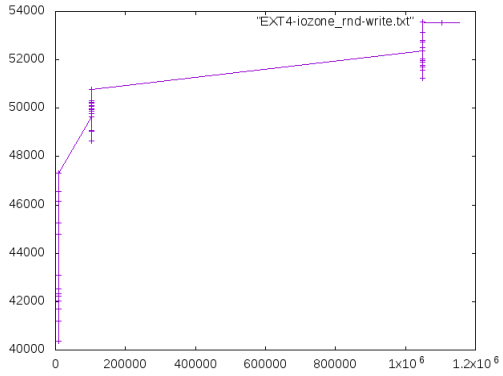
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória

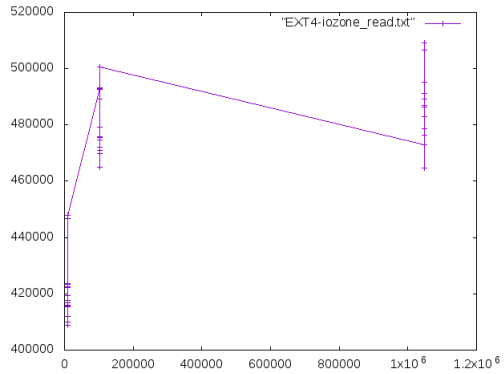


Fonte: Elaborado pelo autor.

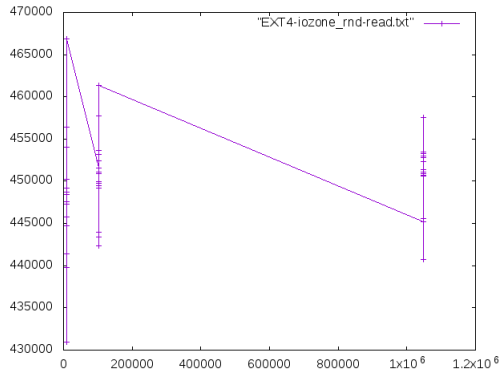
Tabela 29 – Desempenho do EXT4 em um SSD

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do EXT4 (KiByte/s)

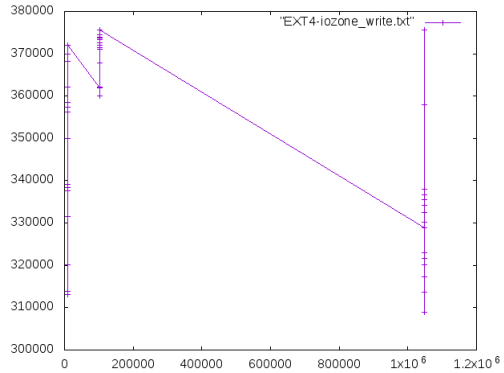
(a) Leitura sequencial



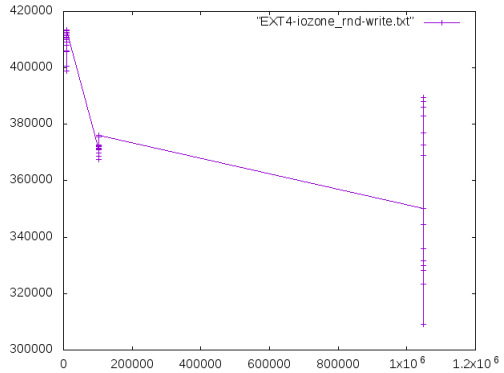
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória

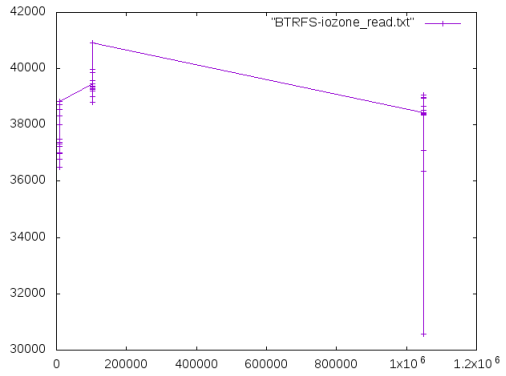


Fonte: Elaborado pelo autor.

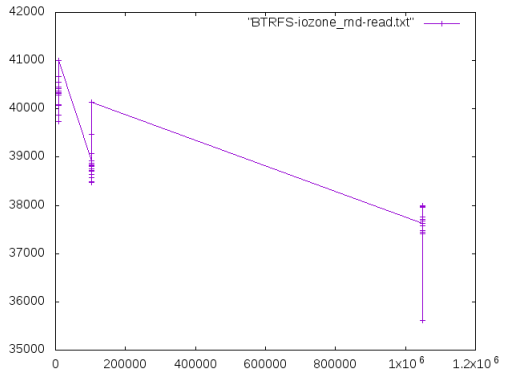
Tabela 30 – Desempenho do BTRFS em um *PenDrive*

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do BTRFS (KiByte/s)

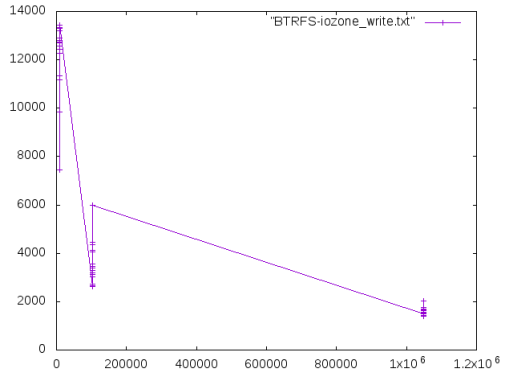
(a) Leitura sequencial



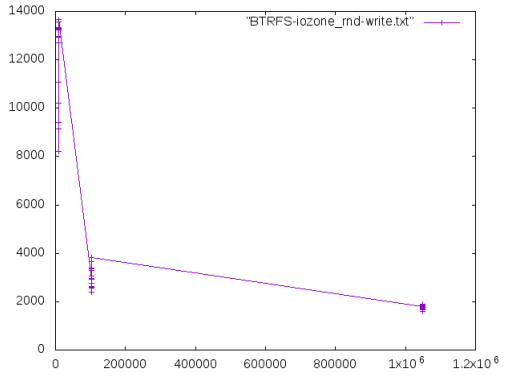
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória

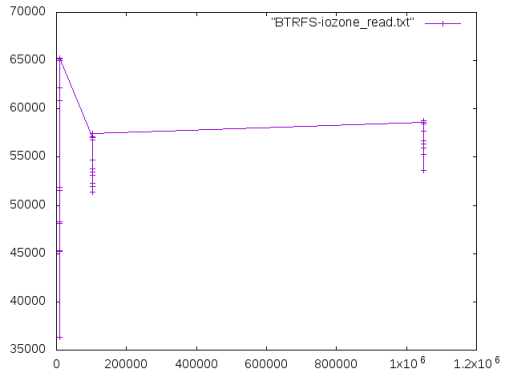


Fonte: Elaborado pelo autor.

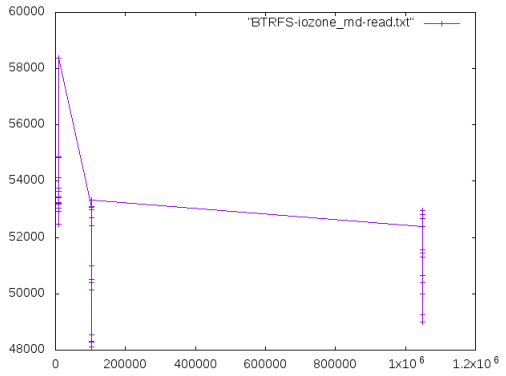
Tabela 31 – Desempenho do BTRFS em um HDD

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do BTRFS (KiByte/s)

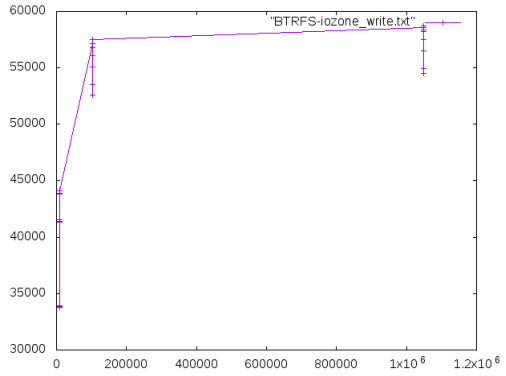
(a) Leitura sequencial



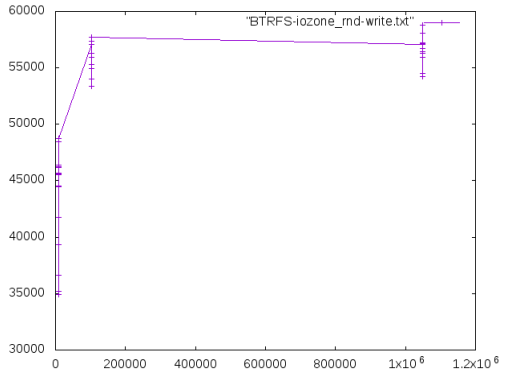
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória

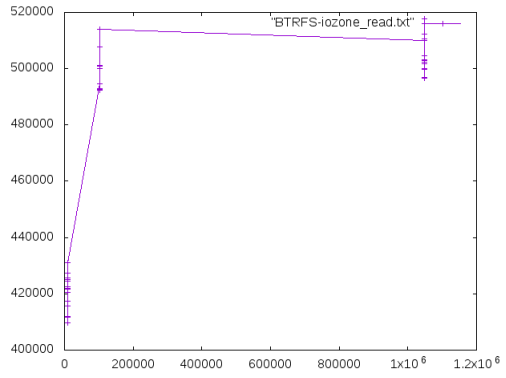


Fonte: Elaborado pelo autor.

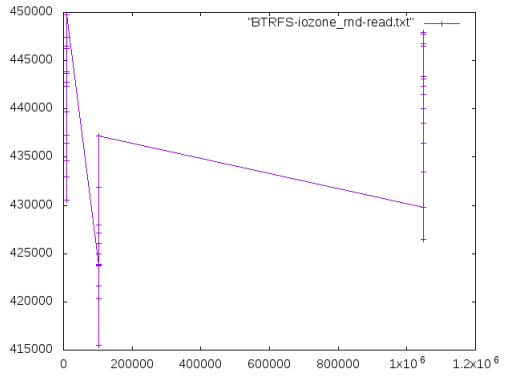
Tabela 32 – Desempenho do BTRFS em um SSD

IOzone *Benchmark* (*File size*: 10 MiB, 100 MiB,1 GiB, *Record length*: 4096 KiB)
Vazão do BTRFS (KiByte/s)

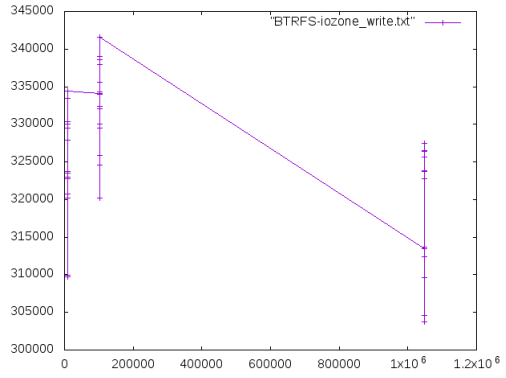
(a) Leitura sequencial



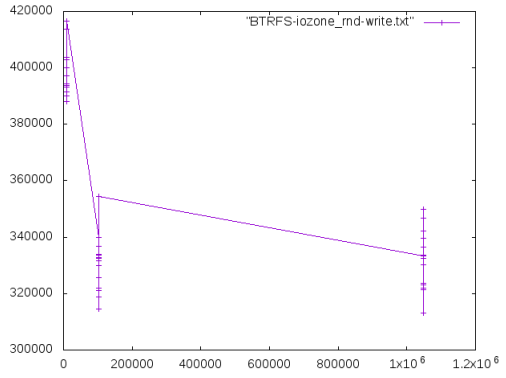
(b) Leitura aleatória



(c) Escrita sequencial



(d) Escrita aleatória



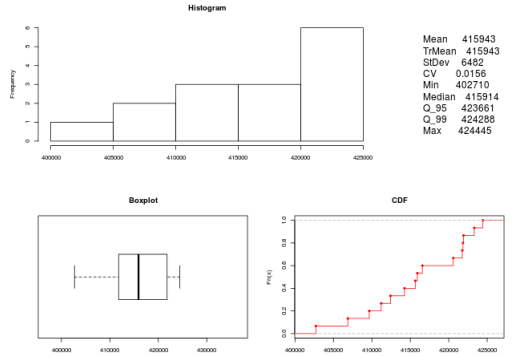
Fonte: Elaborado pelo autor.

APÊNDICE B – TABELAS(VAZÃO DOS FS)

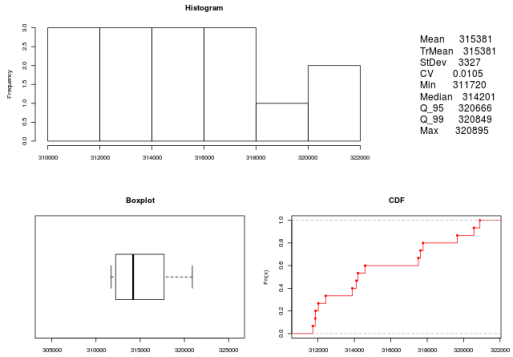
Tabela 33 – Vazão de R/W aleatória dos FS em um SSD

IOzone *Benchmark* (*File size*: 100 MiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

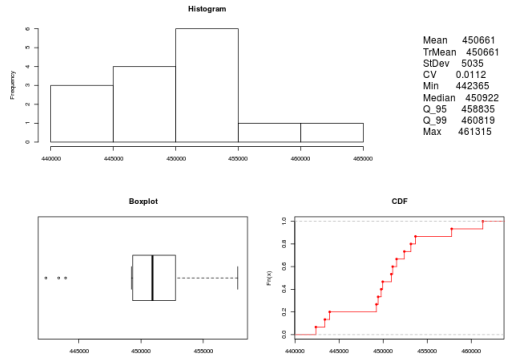
(a) NTFS — Leitura aleatória



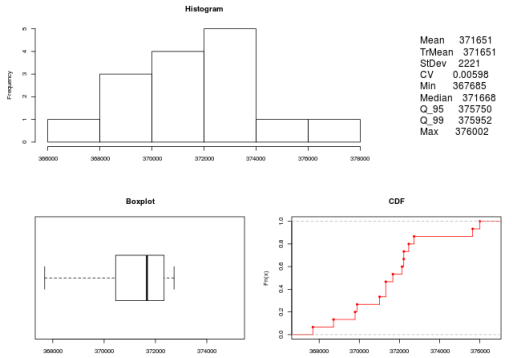
(b) NTFS — Escrita aleatória



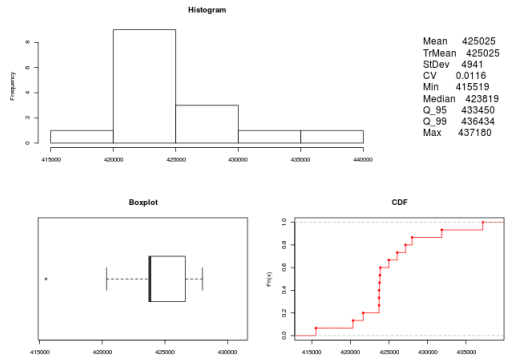
(c) EXT4 — Leitura aleatória



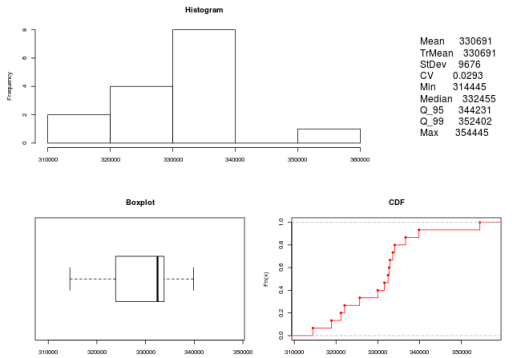
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

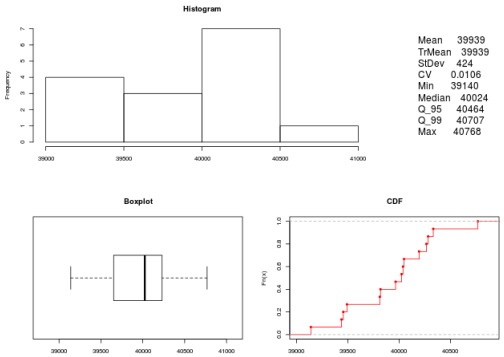
Tabela 34 – Vazão mediana dos FS em um SSD

IOzone *Benchmark* (*File size*: 100MiB, *Record length*: 4096 KiB) Vazão (KiByte/s) — Mediana						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	329888	314201	0,95	464632	415914	0,89
EXT4	371942	371668	0,99	475710	450922	0,94
BTRFS	334002	332455	0,99	492317	423819	0,86

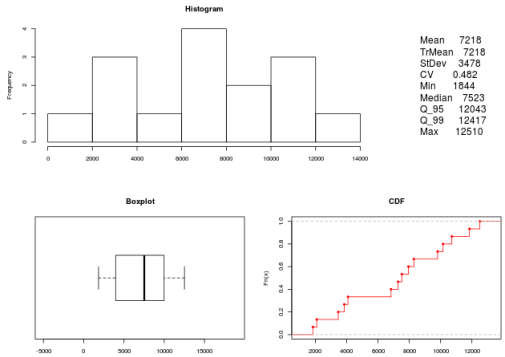
Tabela 35 – Vazão de R/W aleatória dos FS em um *Flash Drive*

IOzone *Benchmark* (*File size*: 10 MiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

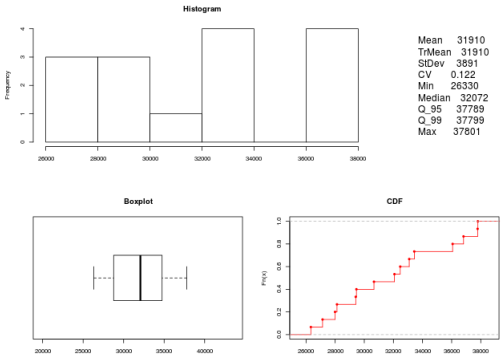
(a) NTFS — Leitura aleatória



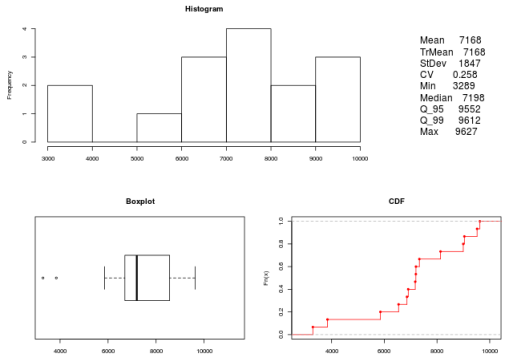
(b) NTFS — Escrita aleatória



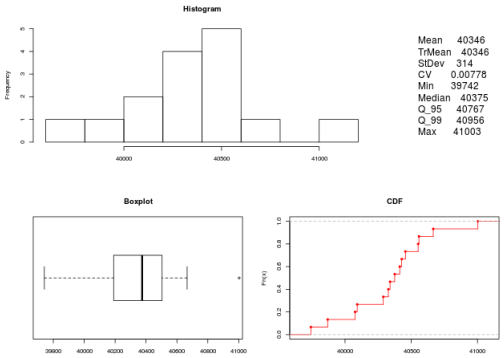
(c) EXT4 — Leitura aleatória



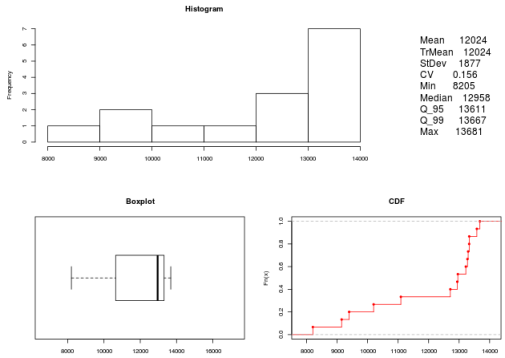
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

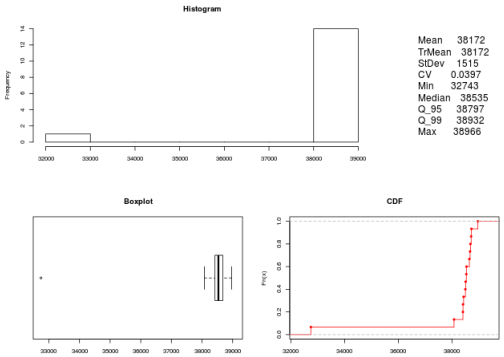
Tabela 36 – Vazão mediana dos FS em um *Flashdrive*

IOzone *Benchmark* (*File size*: 10MiB, *Record length*: 4096 KiB) Vazão (KiByte/s) — Mediana						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	11868	7523	0,63	32860	40024	1,21
EXT4	6565	7198	1,09	30456	32072	1,05
BTRFS	12715	12958	1,01	37395	40375	1,07

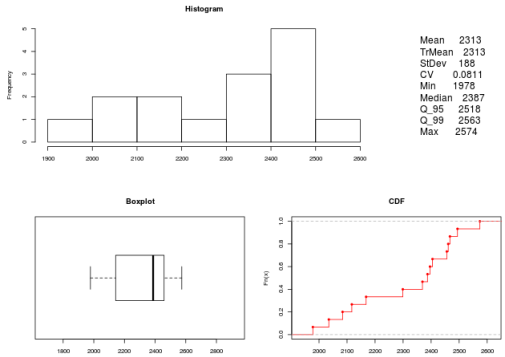
Tabela 37 – Vazão de R/W aleatória dos FS em um *Flash Drive*

IOzone *Benchmark* (*File size*: 100 MiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

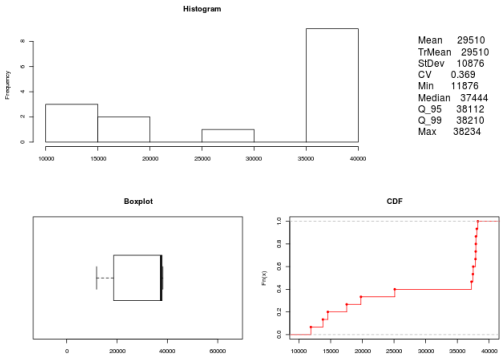
(a) NTFS — Leitura aleatória



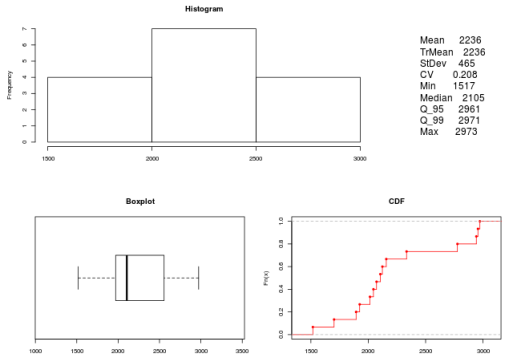
(b) NTFS — Escrita aleatória



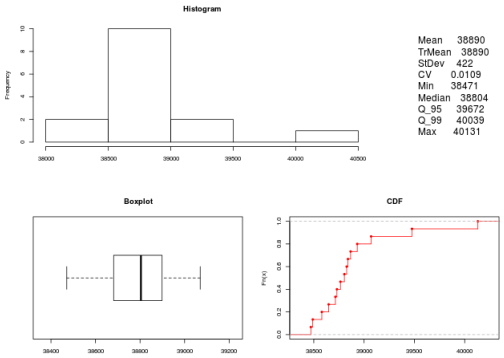
(c) EXT4 — Leitura aleatória



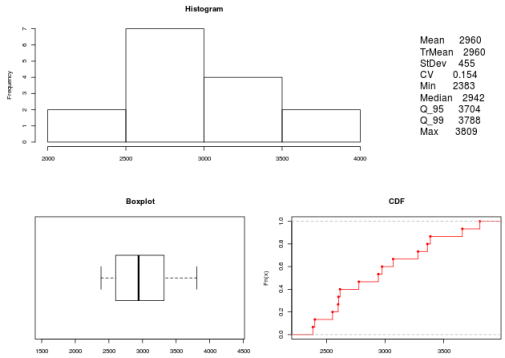
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

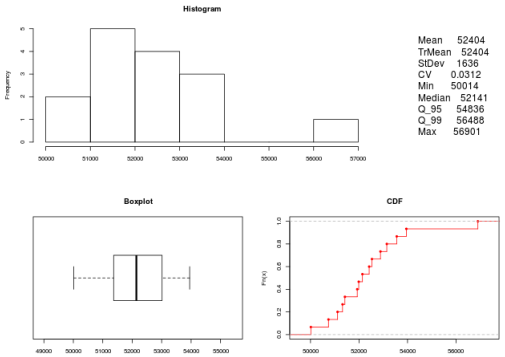
Tabela 38 – Vazão mediana dos FS em um *Flashdrive*

IOzone *Benchmark* (*File size*: 100MiB, *Record length*: 4096 KiB) Vazão (KiByte/s) — Mediana						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	2600	2387	0,91	38422	38535	1,00
EXT4	2133	2105	0,98	32072	37444	1,16
BTRFS	3385	2942	0,86	39373	38804	0,98

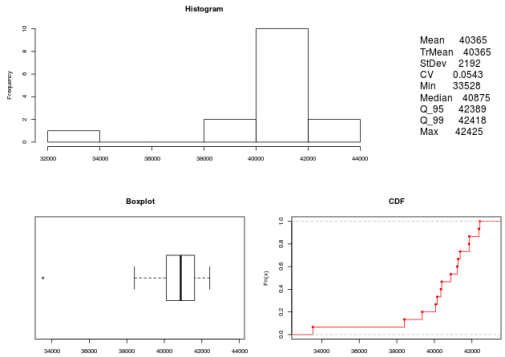
Tabela 39 – Vazão de R/W aleatória dos FS em um HDD

IOzone *Benchmark* (*File size*: 10 MiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

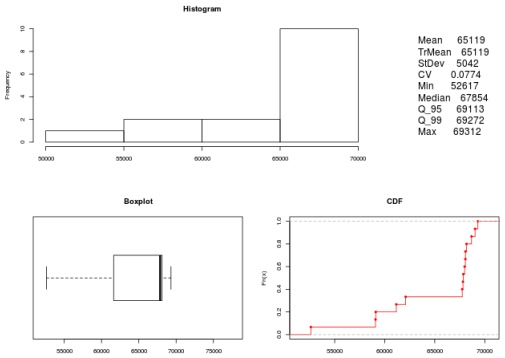
(a) NTFS — Leitura aleatória



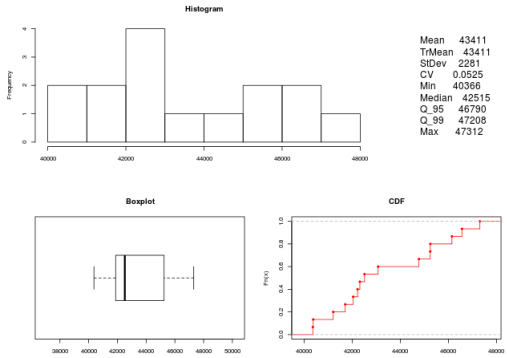
(b) NTFS — Escrita aleatória



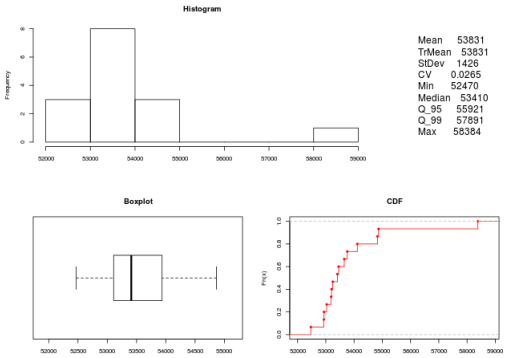
(c) EXT4 — Leitura aleatória



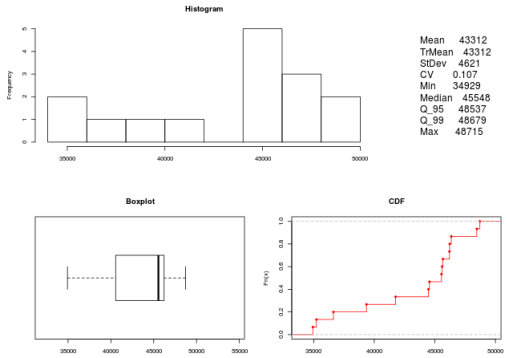
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

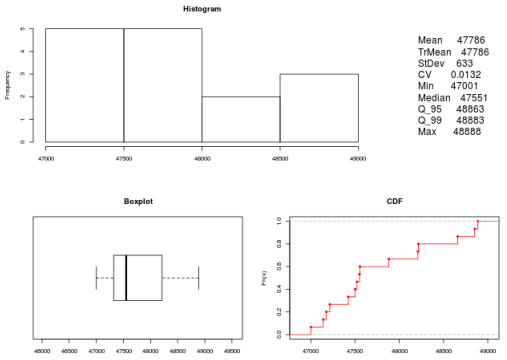
Tabela 40 – Vazão mediana dos FS em um HDD

IOzone *Benchmark* (*File size*: 10MiB, *Record length*: 4096 KiB) Vazão (KiByte/s) — Mediana						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	37181	40875	1,09	39651	52141	1,31
EXT4	41329	42515	1,02	51637	67854	1,31
BTRFS	41407	45548	1,10	48331	53410	1,10

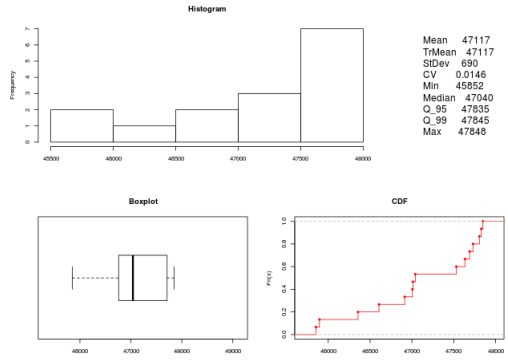
Tabela 41 – Vazão de R/W aleatória dos FS em um HDD

IOzone *Benchmark* (*File size*: 100 MiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

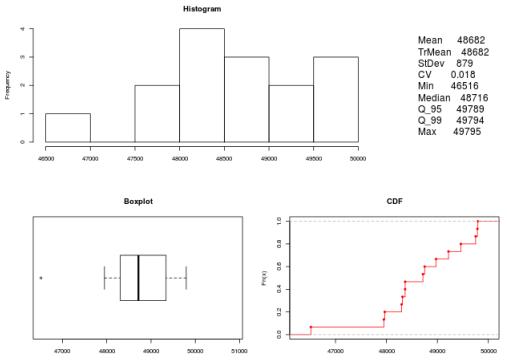
(a) NTFS — Leitura aleatória



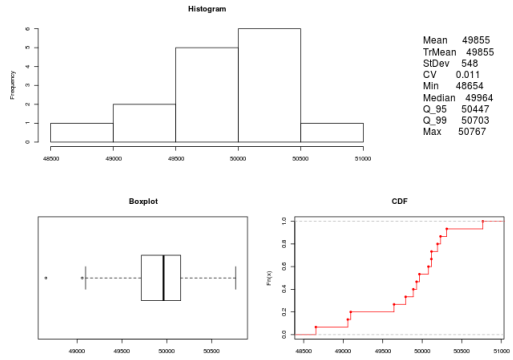
(b) NTFS — Escrita aleatória



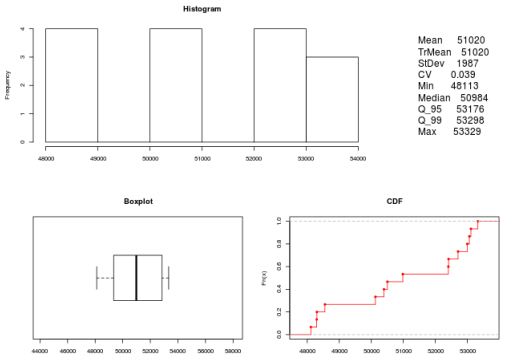
(c) EXT4 — Leitura aleatória



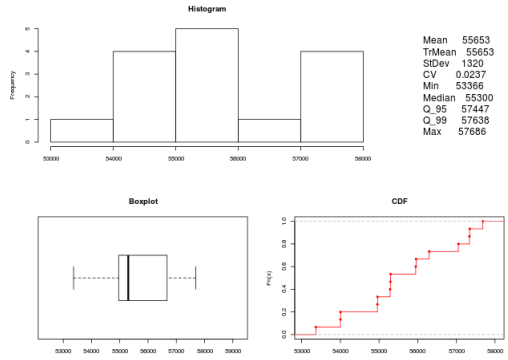
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

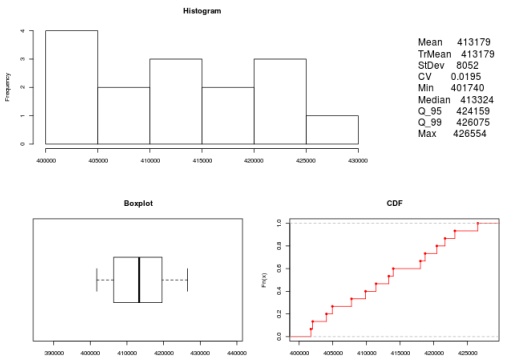
Tabela 42 – Vazão mediana dos FS em um HDD

IOzone *Benchmark* (*File size*: 100MiB, *Record length*: 4096 KiB) Vazão (KiByte/s) — mediana						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	52795	47040	0,89	56085	47551	0,84
EXT4	55085	49964	0,90	56153	48716	0,86
BTRFS	57113	55300	0,96	56743	50984	0,89

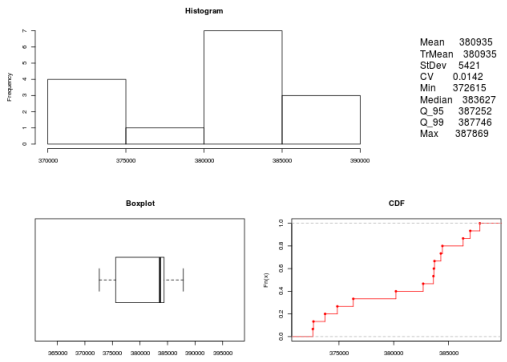
Tabela 43 – Vazão de R/W aleatória dos FS em um SSD

IOzone *Benchmark* (*File size*: 10 MiB, *Record length*: 4096 KiB)
Vazão (KiByte/s)

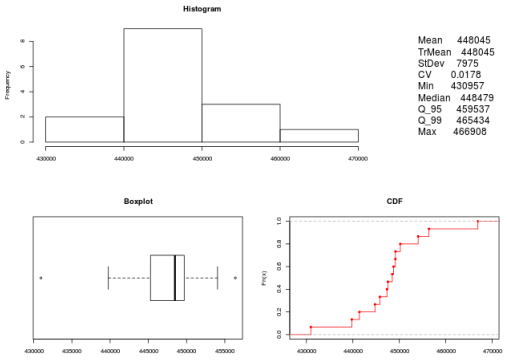
(a) NTFS — Leitura aleatória



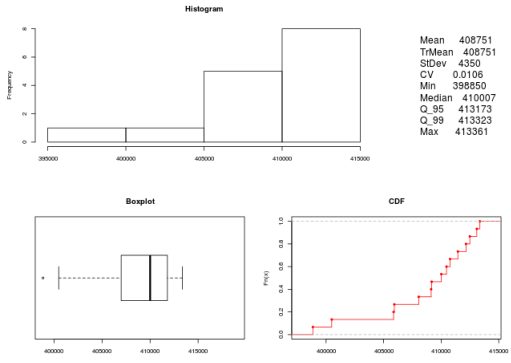
(b) NTFS — Escrita aleatória



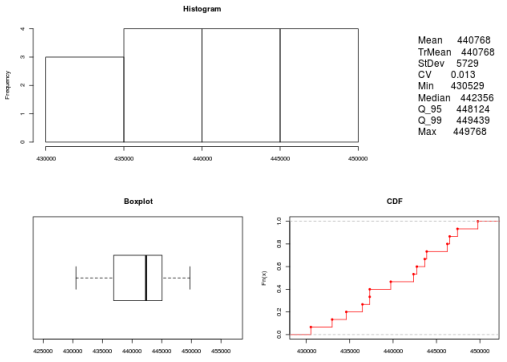
(c) EXT4 — Leitura aleatória



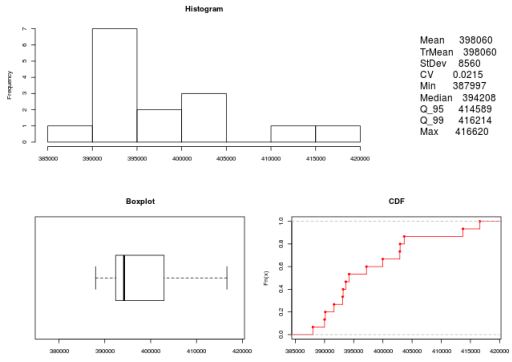
(d) EXT4 — Escrita aleatória



(e) BTRFS — Leitura aleatória



(f) BTRFS — Escrita aleatória



Fonte: Elaborado pelo autor.

Tabela 44 – Vazão mediana dos FS em um SSD

IOzone *Benchmark* (*File size*: 10MiB, *Record length*: 4096 KiB) Vazão (KiByte/s) — Mediana						
FileSystem	Write	Random Write	Razão rnd_W / W	Read	Random Read	Razão rnd_R / R
NTFS	307462	383627	1,24	398912	413324	1,03
EXT4	350054	410007	1,17	417723	448479	1,07
BTRFS	323695	394208	1,21	421744	442356	1,04