UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

DESENVOLVIMENTO DE UM SISTEMA DE ARQUIVOS INSTALÁVEL PARA LINUX

THIAGO KLEIN FLACH

BLUMENAU
2009

2009/2-23

THIAGO KLEIN FLACH

DESENVOLVIMENTO DE UM SISTEMA DE ARQUIVOS INSTALÁVEL PARA LINUX

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos, Orientador

BLUMENAU
2009

2009/2-23

DESENVOLVIMENTO DE UM SISTEMA DE ARQUIVOS INSTALÁVEL PARA LINUX

Por

THIAGO KLEIN FLACH

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Mauro Marcelo Mattos, Orientador, Dr. Eng. – FURB
Membro:	Prof. Antonio Carlos Tavares, MSc – FURB
Membro:	Prof. Francisco Adell Péricas, MSc – FURB

Blumenau, 15 de dezembro de 2009

Dedico este trabalho à minha família e à minha namorada, que me deram todo o apoio necessário durante os meus anos de estudo e especialmente na realização deste trabalho.

AGRADECIMENTOS

A Deus, pelo seu imenso amor.

À minha família, que mesmo longe, sempre esteve presente e me deu muito apoio.

À Aline, minha namorada, que esteve comigo durante meus momentos mais difíceis.

Aos meus amigos, pelas palavras de incentivo.

À Senior Sistemas pela flexibilidade e auxílio em meus estudos.

Ao meu orientador, Mauro Marcelo Mattos, por ter acreditado neste trabalho.

A todo o corpo docente que participou direta ou indiretamente da minha formação.

Determinação, coragem e autoconfiança são fatores decisivos para o sucesso. Se estamos possuídos por uma inabalável determinação conseguiremos superá-los. Independentemente das circunstâncias, devemos ser sempre humildes, recatados e despidos de orgulho.

Dalai Lama

RESUMO

O presente trabalho descreve o projeto de um sistema de arquivos instalável escrito em Java. O sistema foi implementado utilizando como base a plataforma *File System in User Space* (FUSE) e testado em ambiente Linux.

Palavras-chave: Sistemas de arquivos instaláveis. Sistemas de arquivos no espaço do usuário. Linux.

ABSTRACT

This work describes a installable file system written in Java language. The system was implemented on the File System in User Space (FUSE) platform and tested in Linux environment.

Key-words: Installable file systems. File system in user space. Linux.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

*Blogger*FS – *Blogger File System*

ClamAV – Clam *Anti Virus*

ClamFS – Clam *File System*

FUSE – *File system in USEr space*

FUSE-J – *File system in USEr space Java api*

GLIBC – Gnu *LIBrary in* C

IDE – *Integrated Development Environment*

JDK – Java *Development Kit*

JNI – Java *Native Interface*

JVM – Java *Virtual Machine*

MOOFS – *Mobile Objects Oriented File System*

TAR – *Tape ARchive*

TKFFS – Thiago Klein Flach *File System*

VFS – *Virtual File System*

SUMÁRIO

1 INTRODUÇÃO

Todo programa de aplicação, via de regra, precisa armazenar, recuperar e eventualmente compartilhar dados com outros programas. A forma como isso ocorre é fortemente dependente de como o sistema operacional organiza e mantém essas informações nos chamados meios de armazenamento secundário.

Segundo Machado e Maia (2007, p. 214), 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.

O sistema de arquivos é a parte mais visível de um sistema operacional, pois a manipulação de arquivos é uma atividade freqüentemente realizada pelos usuários. Essa manipulação sempre ocorre de maneira uniforme, independendo dos diferentes dispositivos de armazenamento.

> 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, 2000, p. 8).

Love (2004, p. 201) comenta que o *Kernel*[1] do Linux possui um subsistema chamado VFS (*Virtual File System*), que possibilita uma interface comum e transparente ao usuário, independente do sistema de arquivos utilizado.

Segundo Jones (2007), há muitos sistemas de arquivos e mídias. Com toda essa variedade é possível esperar que a interface do sistema de arquivos Linux seja implementada como uma arquitetura de camadas, separando a camada da interface com o usuário da implementação do sistema de arquivos e dos *drivers* que manipulam os dispositivos de armazenamento.

> A arquitetura do sistema de arquivos Linux é um exemplo interessante de complexidade e abstração. Usando um conjunto comum de funções da API, uma grande variedade de sistema [sic] de arquivos pode ter suporte em uma grande variedade de dispositivos de armazenamento. (JONES, 2007).

O presente trabalho tem como intuito o desenvolvimento de um novo sistema de arquivos instalável em distribuições Linux, que trabalhe com um dispositivo de armazenamento específico, que possa ser transportado e utilizado em outras máquinas sem perda do conteúdo.

---

[1] A tradução literal de *kernel* é núcleo, portanto *Kernel* é o núcleo do sistema operacional.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um sistema de arquivos instalável para distribuições Linux.

Os objetivos específicos do trabalho são:

a) estudar e descrever aspectos estruturais de um sistema de arquivos virtual;

b) desenvolver um protótipo de validação de um sistema de arquivos virtual.

1.2 ESTRUTURA DO TRABALHO

O primeiro capítulo apresenta uma introdução do trabalho, seus objetivos e sua estrutura.

O segundo capítulo contempla a fundamentação teórica, descrevendo os conceitos de sistemas de arquivos, VFS, sistemas de arquivo no espaço do usuário, que inclui a plataforma apresentada em Fuse (2009) e o FUSE-J (*File System in User Space Java API*), além de apresentar alguns trabalhos correlatos.

No terceiro capítulo é apresentado o desenvolvimento de um sistema de arquivos em Java, utilizando a interface FUSE-J e executando sobre a plataforma FUSE (*File System in User Space*).

O quarto capítulo apresenta as considerações finais do trabalho, além de sugestões para extensões em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma introdução aos conceitos de sistemas de arquivos e VFS, sistema de arquivos no espaço do usuário, que inclui a plataforma FUSE e a API (*Application Programming Interface*) FUSE-J, além de alguns trabalhos correlatos.

2.1 SISTEMAS DE ARQUIVOS

Jones (2007) define um sistema de arquivos como sendo uma organização de dados e metadados em um dispositivo de armazenamento.

Já Tanenbaum e Woodhull (2000, p. 27) dizem que o sistema de arquivos é a forma com a qual o sistema operacional esconde as peculiaridades dos discos e outros dispositivos de entrada e saída, apresentando um modelo abstrato de arquivos independente de dispositivos. É nos sistemas de arquivos que são implementadas chamadas para operações como: criar, remover, ler e escrever arquivos.

Esses sistemas atuam como uma interface entre os dados (arquivos) e o usuário, através da qual o usuário pode interagir com esses dados, manipulando-os conforme desejar.

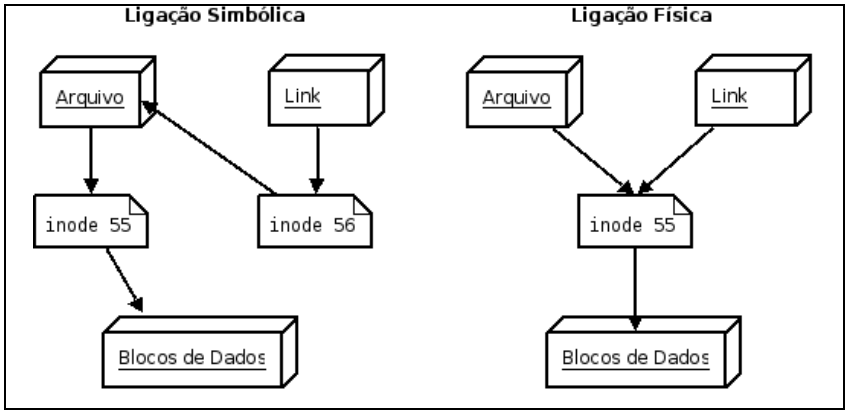
> Um sistema de arquivos é um armazenamento hierárquico de dados que segue uma estrutura específica. Os sistemas de arquivos contêm arquivos, diretórios e informações de controle associadas. As operações típicas executadas nos sistemas de arquivos são a criação, a eliminação e a montagem. No Unix, os sistemas de arquivos são montados em um ponto de montagem específico em uma hierarquia global conhecida como *espaço do nome*. Isso permite que todos os sistemas de arquivos montados apareçam como entradas em uma única árvore. (LOVE, 2004, p. 203, grifo do autor).

Jones (2007) 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. 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.

Sintetizando o que foi dito por Love (2004, p. 203), os sistemas Unix separam o conceito de arquivo de qualquer informação relacionada a ele (permissões de acesso, tamanho, proprietário, entre outros). Essas informações normalmente são chamadas de

metadados e são armazenadas em uma estrutura de dados separada do arquivo, chamada de `inode` (*index node*).

Na Figura 1 é possível observar como é realizado o acesso aos dados de um arquivo no disco, em um sistema Unix. 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.



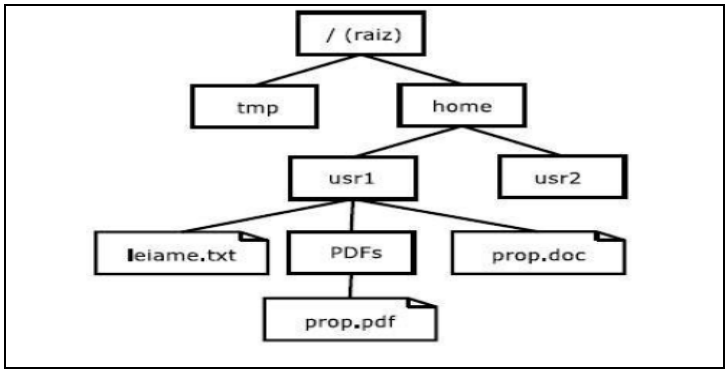
Fonte: Leite (2008).

Figura 1 – Exemplo do acesso dos dados de um arquivo no disco

Deitel, Deitel e Choffnes (2005, p. 626) ainda complementam que em Unix, arquivos servem como pontos de acesso a dados, que podem ser encontrados em discos locais, na rede ou até mesmo gerados pelo *Kernel*, abstraindo assim o conceito de arquivo.

Uma estrutura hierárquica busca organizar informações em uma ordem lógica ou de importância. Normalmente apresenta uma estrutura de árvore onde cada nó da estrutura tem apenas um correspondente superior. Na gerência de arquivos, a estrutura hierárquica é geralmente utilizada e nela são os diretórios e arquivos que representam os elementos da estrutura. (TANENBAUM; WOODHULL, 2000, p. 27).

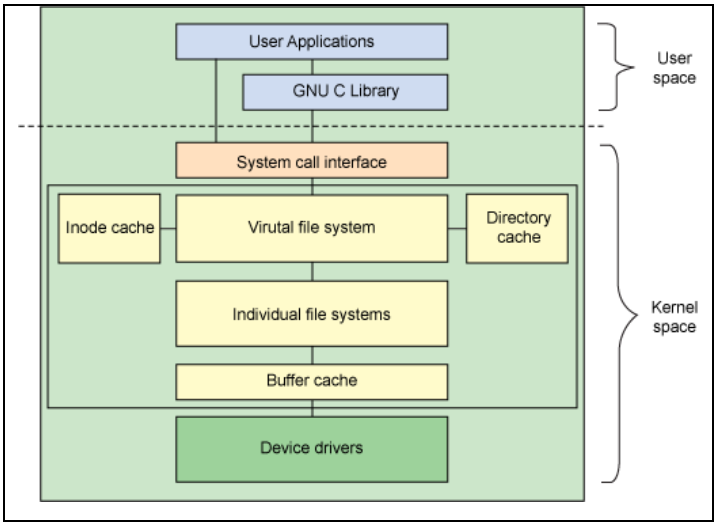
A Figura 2 apresenta um exemplo de uma árvore de diretórios do Linux, para facilitar a compreensão da estrutura hierárquica de um sistema de arquivos. No presente trabalho, a denominação *node* é dada aos nós de uma árvore de diretórios.



Fonte: Carvalho (2005, p. 3).

Figura 2 – Exemplo de uma árvore de diretórios

Para Jones (2007), os relacionamentos entre os componentes do sistema de arquivos principal dividem-se entre operações no espaço do *Kernel* e operações no espaço do usuário, conforme a Figura 3.



Fonte: Jones (2007).

Figura 3 – Visualização de arquitetura dos componentes do sistema de arquivos Linux

> O espaço do usuário contém aplicativos [...] e o GNU C *Library* (glibc), que fornece a interface do usuário para as chamadas do sistema de arquivos (abrir, ler, gravar e fechar). A interface de chamada do sistema atua como um comutador, canalizando as chamadas do sistema do espaço do usuário para terminais adequados no espaço do *kernel*. (JONES, 2007, grifo nosso).

Jones (2007) ainda descreve a estrutura dos sistemas de arquivos Linux. Ele começa sua descrição dizendo que o Linux visualiza todos esses sistemas da perspectiva de um conjunto de objetos comum. Esses objetos são `superblock`, `inode`, `dentry` e `file`.

> Na raiz de cada sistema de arquivos está superblock, que descreve e mantém o estado do sistema de arquivos. Cada objeto que é gerenciado dentro de um sistema de arquivos (arquivo ou diretório) é representado no Linux como um inode. O inode contém todos os metadados para gerenciar objetos no sistema de arquivos (inclusive as operações que são possíveis nele). Outro conjunto de estruturas, chamadas dentries, é usado para conversão entre nomes e inodes, para o qual um cache de diretório existe para manter por perto o objeto usado mais recentemente. O dentry também mantém relacionamentos entre diretórios e arquivos para sistemas de arquivos desviados. Por fim, um arquivo VFS representa um arquivo aberto [...]. (JONES, 2007).

Em resumo ao já dito por Jones (2007), as estruturas do sistema de arquivos Linux são definidas da seguinte forma:

a) `vfs`: atua como o nível raiz da interface do sistema, mantendo o rastreamento dos sistemas de arquivos suportados atualmente, bem como dos que estão sendo montados no momento;
b) `superblock`: estrutura que representa um sistema de arquivos. Essa estrutura armazena as informações necessárias para gerenciar o sistema durante a operação. Dentre essas informações estão o nome do sistema de arquivos, seu tamanho, seu

estado, uma referência para o dispositivo de bloco. Essa estrutura geralmente é armazenada na mídia de armazenamento, mas pode ser criada em tempo real se não existir nenhuma;

c) `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;

d) `dentry`: estruturas utilizadas na conversão entre nomes de arquivos e `inodes`;

e) `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.

Pfenning (2009, tradução nossa) afirma: “Sistemas de arquivos provêm uma abstração, tanto para o usuário de um sistema computacional, quanto para o programador. Eles apresentam uma visão uniforme e hierárquica de dados, mesmo que esses dados possam estar distribuídos por várias áreas do disco, ou possam estar espalhados por vários discos ou sistemas computacionais”.

Segundo Deitel, Deitel e Choffnes (2005, p. 626), para que isso seja possível foi criada uma interface genérica entre o *Kernel* e os sistemas de arquivos, chamada VFS. Com ela, é possível desenvolver novos sistemas e integrá-los ao sistema operacional de uma forma mais abstrata.

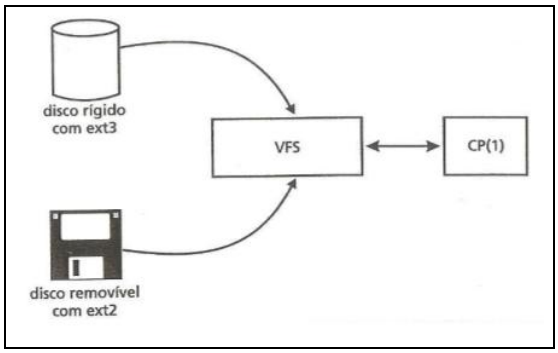
2.2 VIRTUAL FILE SYSTEM (VFS)

Como visto anteriormente, o Unix suporta vários sistemas de arquivos por meio de uma camada de sistema de arquivo virtual, chamada *Virtual File System* ou simplesmente VFS.

> O VFS abstrai os detalhes de acesso ao arquivo permitindo que usuários vejam todos os arquivos e diretórios do sistema sob uma única árvore de diretório. Usuários podem acessar qualquer arquivo na árvore de diretório sem saber onde, e [sic] sob qual sistema de arquivo, [sic] os dados do arquivo estão armazenados. Todas as requisições relacionadas ao arquivo são inicialmente enviadas à camada VFS que fornece uma interface para acessar dados de arquivo em qualquer sistema de

> arquivos disponível. O VFS oferece apenas uma definição básica dos objetos que compreendem um sistema de arquivo [sic]. Sistemas de arquivos individuais expandem aquela definição básica para incluir detalhes sobre o modo como objetos são armazenados e acessados. (DEITEL; DEITEL; CHOFFNES, 2005, p. 626).

A Figura 4 exemplifica um processo de uma cópia sendo feita de um arquivo presente em um disco rígido para um disco removível.



Fonte: Love (2004, p. 201).

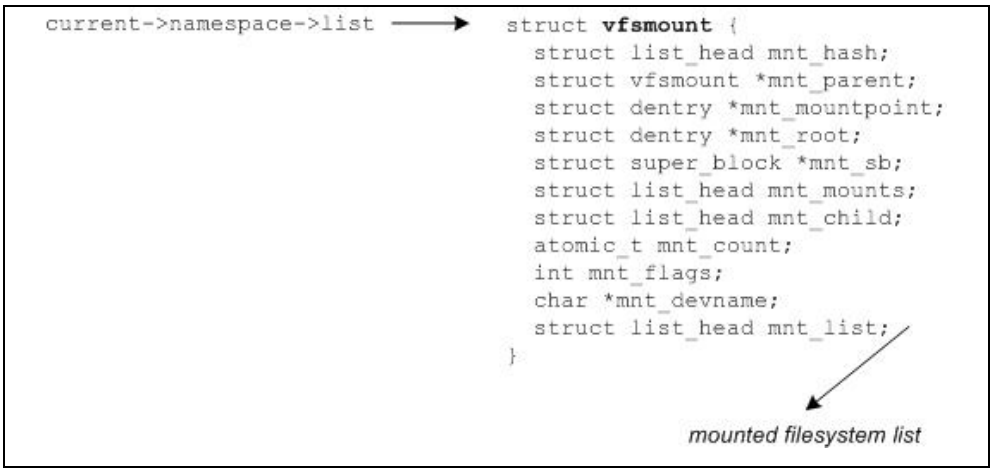
Figura 4 – Exemplo de cópia de arquivo entre dois dispositivos através do VFS

Para Jones (2007), o VFS é a interface primária para os sistemas de arquivos subjacentes. Ele exporta um conjunto de interfaces e depois as concentra nos sistemas de arquivos individuais, que podem ter comportamentos distintos. Cada implementação de sistema de arquivos exporta um conjunto comum de interfaces (objetos) que é usado e esperado pelo VFS.

Jones (2007) ainda comenta que o VFS é responsável por manter as estruturas referentes ao registro de sistemas de arquivos no Linux (Figura 5), que podem ser utilizadas na hora da sua adição ou remoção dinâmica no sistema operacional. Outra estrutura que o VFS mantém é o sistema de arquivos montado (Figura 6), que é vinculada a estrutura `superblock` e que fornece os sistemas de arquivos que estão montados atualmente.

Fonte: Jones (2007).

Figura 5 – Estrutura referente aos sistemas de arquivos registrados com o *Kernel*



Fonte: Jones (2007).

Figura 6 – Estrutura referente à lista de sistemas de arquivos montados

Segundo Love (2004, p. 201), o VFS é um subsistema do *Kernel* que implementa a interface do sistema de arquivos fornecida para os programas do espaço do usuário. É através dele que esses sistemas coexistem, interoperam, o que permite o uso de chamadas padrão do sistema operacional para ler e gravar em sistemas de arquivos diferentes em meios diferentes.

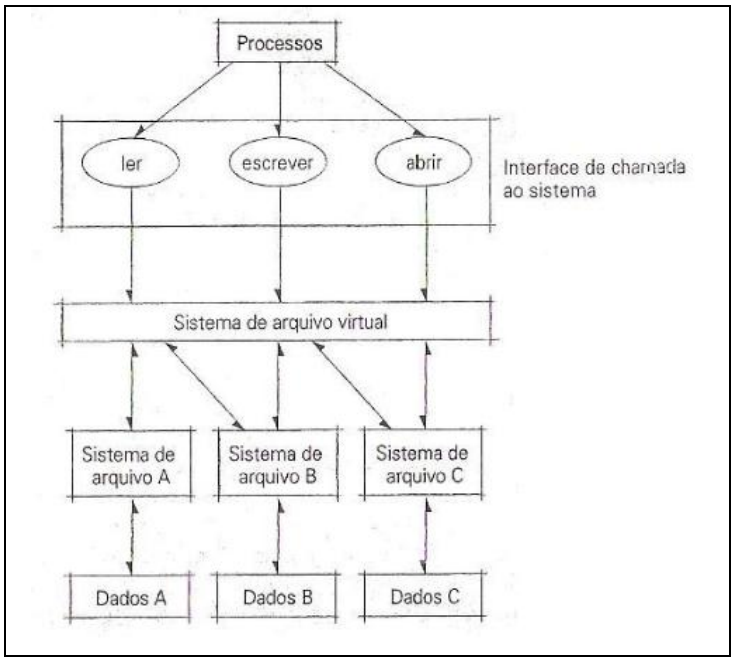
O Quadro 1 apresenta algumas das principais funções disponibilizadas pelo VFS para se trabalhar com arquivos e diretórios.

Função	Descrição
`llseek`	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 loga-o 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: adaptado de Love (2004, p. 221-223).

Quadro 1 – Principais funções do VFS para manipular arquivos e diretórios

É o VFS quem permite aos sistemas de arquivos, efetuarem chamadas como `open`, `read` e `write` sem que tenham que implementar em si 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 si, como na Figura 7.



Fonte: Deitel, Deitel e Choffnes (2005, p. 626).

Figura 7 – Exemplo diagramado de procedimentos realizados em sistemas de arquivos Linux

2.2.1 Camada de abstração do sistema de arquivos

Segundo Love (2004, p. 202), o próprio *Kernel* do sistema operacional possui uma camada de abstração que o permite suportar diferentes sistemas de arquivos, mesmo se eles diferirem muito nos recursos suportados ou comportamento. Para isso ser possível o VFS fornece um modelo de arquivo comum que é capaz de representar qualquer recurso e comportamento gerais do sistema de arquivos concebível. A camada de abstração funciona definindo as interfaces abstratas básicas e as estruturas de dados que todos esses sistemas suportam. Cada um deles molda sua visão dos conceitos para coincidirem com as expectativas do VFS. Isso faz com que para a camada VFS e o resto do *Kernel*, cada sistema de arquivos pareça igual, suportando noções, como arquivos e diretórios e operações, como criar e apagar um arquivo.

Um exemplo de estrutura utilizada pelo VFS para fornecer um objeto compatível com a camada de abstração do sistema operacional pode ser visto no Quadro 2.

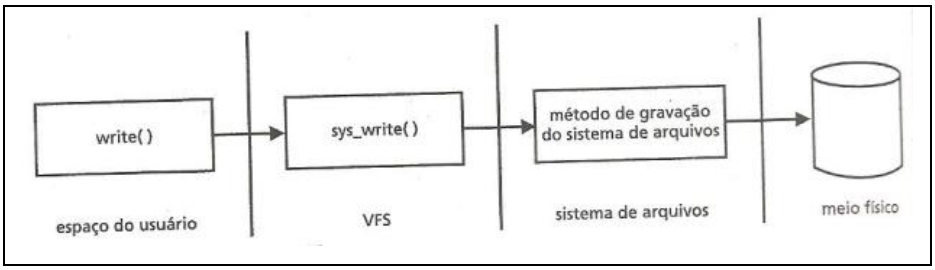
> Os sistemas de arquivos são programados para fornecer as interfaces abstraídas e as estruturas de dados que o VFS espera; por sua vez, o *kernel* trabalha facilmente com qualquer sistema de arquivos, e [sic] a interface do usuário exportada funciona uniformemente em qualquer sistema de arquivos. (LOVE, 2004, p. 202, grifo nosso).

```
struct file {
  struct list_head        f_list;       /* list of file objects */
  struct dentry           *f_dentry;    /* associated dentry object */
  struct vfsmount         *f_vfsmnt;    /* associated mounted
                                           filesystem */
  struct file_operations  *f_op;        /* file operations table */
  atomic_t                f_count;      /* file object's usage count */
  unsigned int            f_flags;      /* flags specified on open */
  mode_t                  f_mode;       /* file access mode */
  loff_t                  f_pos;        /* file offset (file pointer) */
  struct fown_struct      f_owner;      /* owner data for signals */
  unsigned int            f_uid;        /* user's UID */
  unsigned int            f_gid;        /* user's GID */
  int                     f_error;      /* error code */
  struct file_ra_state    f_ra;         /* read-ahead state */
  unsigned long           f_version;    /* version number */
  void                    *f_security;  /* security module */
  void                    *private_data; /* tty driver hook */
  struct list_head        f_ep_links;   /* list of eventpoll links */
  spinlock_t              f_ep_lock;    /* eventpoll lock */
}
```

Fonte: Love (2004, p. 219).

Quadro 2 – Exemplo de estrutura de arquivo no VFS

A Figura 8 apresenta o fluxo de uma chamada `write` do espaço do usuário até o meio físico, através do VFS e da camada de abstração do sistema operacional.



Fonte: Love (2004, p. 204).

Figura 8 – Exemplo do fluxo de uma chamada do espaço do usuário até o meio físico

Love (2004, p. 204) acrescenta que o VFS trabalha com objetos. Um desses objetos (estruturas) é utilizado para representar o modelo de arquivo comum e este modelo de arquivo comum é que representa tanto os diretórios quanto os arquivos em si, o que os difere são as operações executadas sobre eles.

2.3 SISTEMA DE ARQUIVOS NO ESPAÇO DO USUÁRIO

Este capítulo apresenta a plataforma desenvolvida por Fuse (2009) para possibilitar a execução de sistemas de arquivos no espaço do usuário e a interface desenvolvida por Fuse-j (2009) para possibilitar o desenvolvimento desses sistemas que executam sobre a primeira, na linguagem Java.

2.3.1 FILE SYSTEM IN USER SPACE (FUSE)

"Com o FUSE é possível implementar um sistema totalmente funcional como um programa do espaço do usuário" (FUSE, 2009, tradução nossa).

Para Omake (2009), FUSE é um módulo que torna possível para um usuário Unix implementar de forma mais fácil um sistema de arquivos e então montá-lo e usá-lo.

Sintetizando, ele é um módulo que pode ser instalado no *Kernel* do Linux, para possibilitar o desenvolvimento de sistemas de arquivos em C, que executam no espaço do usuário.

Fuse (2009) indica que as principais vantagens do FUSE são:

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.

Segundo Pfenning (2009), as duas maiores dificuldades encontradas para o desenvolvimento de sistemas de arquivos são:

a) as APIs geralmente são extensas e complexas;
b) sistemas de arquivos trabalham no espaço do *Kernel*, o que torna difícil a depuração e facilita a ocorrência de falhas na máquina.

Ainda para Pfenning(2009), o FUSE resolve a primeira dificuldade facilitando o desenvolvimento através de uma API mais simples e uniforme e também resolve a segunda, pois executa o código do sistema de arquivos no espaço do usuário e não no espaço do *Kernel*.

Omake (2009) complementa informando que os modelos estruturais utilizados pelo FUSE são baseados nos modelos do VFS.

Pfenning (2009) exemplifica o procedimento de execução do código de um sistema de arquivos desenvolvido com FUSE, da seguinte forma:

a) um programa (comando) como `ls`, `mkdir`, entre outros, faz uma chamada para uma

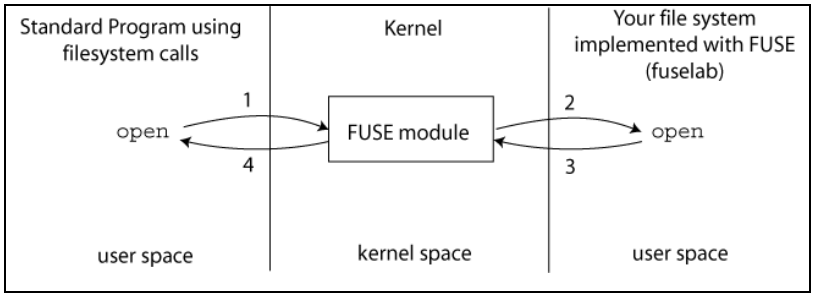
rotina de sistema de arquivos;

b) se esse arquivo está em um volume FUSE, o *Kernel*, através do VFS, passa-o para o seu módulo FUSE, que depois o passa para a implementação do sistema de arquivos;

c) então a implementação da operação `open` faz referências às estruturas de dados reais que representam o sistema de arquivos e devolve um identificador do arquivo. Ele inicia um trabalho para ter uma visão concreta dos dados (*bits* armazenados em um disco rígido) e apresentar uma visão abstrata (um sistema de arquivos organizado hierarquicamente);

d) o *Kernel* retorna o resultado da função `open` para o programa que originalmente fez a chamada.

A Figura 9 facilita a compreensão do procedimento de execução de uma operação por um sistema de arquivos desenvolvido para sobre o FUSE.



Fonte: Pfenning (2009).

Figura 9 – Exemplo do procedimento realizado durante a execução de uma operação sobre o FUSE

O Quadro 3 apresenta algumas das principais funções da API do FUSE, tanto para arquivos comuns, quanto para diretórios.

	Arquivo	**Diretório**
criar	`mknod`	`mkdir`
remover	`unlink`	`rmdir`
ler	`read`	`readdir`
escrever	`write`	
outros	`open, truncate`	

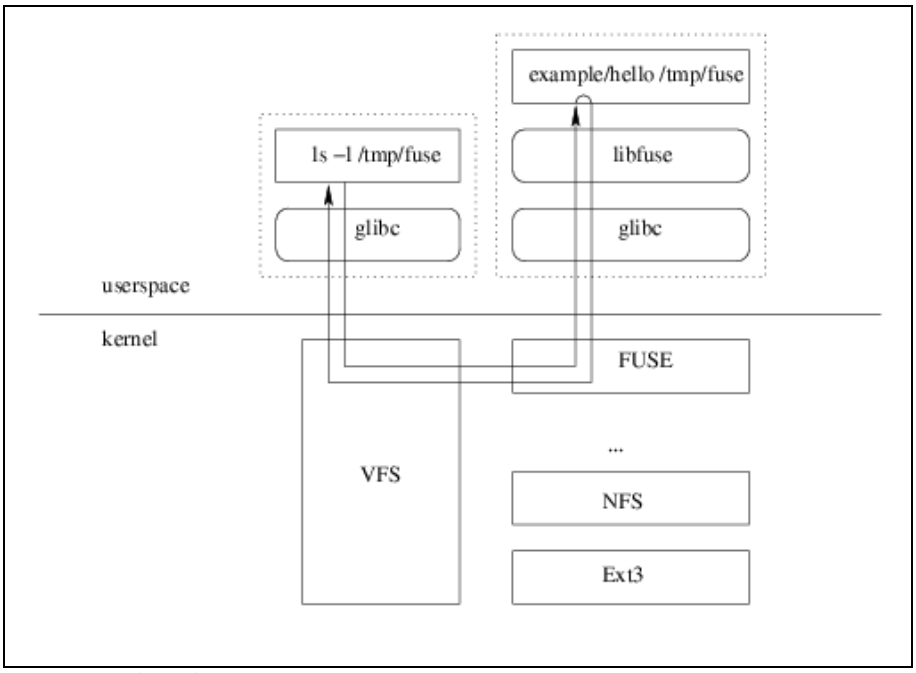
Fonte: adaptado de Pfenning (2009).

Quadro 3 – Funções do FUSE para manipular arquivos comuns e diretórios

Após desenvolver um sistema de arquivos para rodar sobre o FUSE e este for devidamente compilado, o usuário necessitará mapear o seu sistema em algum diretório para poder usar. O fluxo de uma chamada de sistema utilizando esta plataforma pode ser entendido pelo seguinte exemplo (Figura 10):

a) o usuário executa um comando dentro do diretório onde o sistema de arquivos desenvolvido está mapeado;

b) o GLIBC (GNU *Library in* C) interpreta o comando e envia uma requisição ao VFS, que está no espaço do *Kernel*.
c) ao receber a requisição, o VFS identifica que a requisição é referente a um sistema de arquivos que executa sobre o FUSE, e envia ao FUSE a requisição;
d) ao receber a requisição, o FUSE utiliza as bibliotecas que estão em seu diretório `/dev/fuse` para executar o código referente ao sistema de arquivos desenvolvido;
e) a partir daí o fluxo se inverte e o sistema de arquivos envia ao usuário as informações necessárias através dos objetos que são trocados entre o FUSE e o VFS.



Fonte: Fuse (2009).
Figura 10 – Exemplo do fluxo de uma chamada de sistema utilizando o FUSE

Jones (2007) complementa dizendo que o FUSE é um projeto interessante, que permite que o roteamento de pedidos do sistema de arquivos, através do VFS, novamente para o espaço do usuário. Isso permite que os tratamentos dados à requisição do sistema de arquivos, sejam realizados por aplicações desenvolvidas para também serem executadas no espaço do usuário.

Omake (2009) entende as funções do FUSE como mais simplificadas quando comparadas às do VFS. Os parâmetros recebidos pelas suas funções são mais simples de manipular, tornando o desenvolvimento do sistema de arquivos mais fácil. O Quadro 4 apresenta a descrição das funções do FUSE.

Função	Descrição
getattr(name : String, stat : Stat)	Obtém os atributos do arquivo especificado. `name` é o nome do arquivo, com relação ao ponto de montagem do sistema de arquivos. O argumento `stat` deve ser preenchido com as informações do arquivo.
readlink(name : String, buffer : char *, len : nativeint)	`name` refere-se ao *link* simbólico. O método deve copiar o alvo do *link* para o `buffer`.
mknod(name : String, mode : nativeint, dev : nativeint)	Cria um arquivo ou dispositivo com o modo (`mode`) e o nome (`name`) determinado.
mkdir(name : String, mode : nativeint)	Cria um diretório com as permissões (`mode`) e o nome (`name`) determinado.
unlink(name : String)	Remove o arquivo especificado.
rmdir(name : String)	Remove o diretório especificado.
symlink(name : String, link : String)	Cria um *link* simbólico. `name` é o nome do *link* e o parâmetro `link` é o seu alvo.
rename(name1 : String, name2 : String)	Renomeia (`mv`) o arquivo especificado.
link(name1 : String, name2 : String)	Cria um *link* físico. `name1` é o nome do arquivo a ser criado. `name2` é o arquivo a que ele se refere.
chmod(name : String, mode : nativeint)	Muda as permissões (`mode`) do arquivo, dispositivo, ou diretório especificado.
chown(name : String, uid : nativeint, gid : nativeint)	Muda o dono (`uid`) e o grupo (`gid`) do arquivo especificado.
truncate(name : String, len : nativeint)	Muda o tamanho (`len`) do arquivo especificado.
utime(name : String, time : Timespec)	Muda a data de modificação (`time`) do arquivo especificado.
open(name : String, info)	Abre o arquivo especificado. Isto é uma otimização, pois torna possível iniciar algumas estruturas de dados quando o arquivo é aberto.
read(name : String, buffer : char *, size : nativeint, off : nativeint)	Lê alguns dados do arquivo especificado. A função deve retornar o número total de *bytes* lidos ou um erro em caso de falha.
write(name : String, buffer : char *, size : nativeint, off : nativeint)	Escreve alguns dados no arquivo especificado. A função deve retornar o número total de *bytes* gravados ou um erro em caso de falha.
statfs(name : String, stat : Statfs)	Busca as informações do arquivo especificado.
flush(name : String, info)	Limpa o arquivo especificado no disco. Esta é uma customização.
release(name : String, info)	É executado quando um processo libera o arquivo especificado. Isto não significa que o arquivo está fechado, pois podem haver outros processos que o abriram.
fsync(name : String, sync, info)	Sincroniza o arquivo especificado, liberando seu conteúdo para armazenamento permanente.
setxattr(name : String, attr : String, buffer : char*, size : nativeint, i : nativeint)	Define um atributo estendido para o arquivo especificado.
getxattr(name : String, attr : String, buffer : char*, size : nativeint)	Obtém um atributo estendido do arquivo especificado.
listxattr(name : String, buffer : char*, size : nativeint)	Lista os nomes de todos os atributos estendidos do arquivo especificado.
removexattr(name : String, attr : String)	Remove um atributo estendido do arquivo especificado.
opendir(name : String, info)	Abre o diretório especificado, preparando-o para leitura.
readdir(name : String, buffer : char*, fill-dir, off : nativeint, info)	Lê a próxima entrada de diretório para o caminho especificado.
releasedir(name : String, info)	É executado quando um processo libera o diretório especificado e não está mais o lendo.
fsyncdir(name : String, sync, info)	Libera o diretório para armazenamento permanente.
init(conn)	Chamado quando o sistema de arquivos é montado.
destroy(data)	Chamado quando o sistema de arquivos é desmontado.
access(name : String, mode : nativeint)	Verifica se o acesso dado pode ser permitido.

create(name : String, mode : nativeint, info)	Cria um arquivo vazio com a permissão (`mode`) especificada.
ftruncate(name : String, off : nativeint, info)	Muda o tamanho do arquivo especificado.
fgetattr(name : String, stat : Stat, info)	Obtém os atributos de status do arquivo especificado.
lock(name : String, info, cmd, lock)	Gerencia os bloqueios de arquivo.
utimens(name : String, tv : Timeval)	Muda os tempos (data e hora) do arquivo especificado.
bmap(name : String, blocksize : nativeint, idx)	Eu acredito que este deve retornar uma máscara de *bits* indicando quais blocos do arquivo estão na verdade alocados no disco.

Fonte: adaptado de Omake (2009).

Quadro 4 – Descrição das funções do FUSE

Quando ocorrem falhas durante a execução das funções, o sistema de arquivos retorna ao FUSE um identificador para as mesmas. Esses identificadores são chamados de `errno`. É conhecendo o `errno` que o FUSE retorna ao VFS um *feedback* sobre sua requisição.

No Quadro 5 Fuse (2009) apresenta as definições dos erros (`errno`) que podem ser retornados pelo FUSE ao VFS.

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define  EPERM     1 /* Operation not permitted */
#define  ENOENT      2 /* No such file or directory */
#define  ESRCH     3 /* No such process */
#define  EINTR     4 /* Interrupted system call */
#define  EIO       5 /* I/O error */
#define  ENXIO     6 /* No such device or address */
#define  E2BIG     7 /* Argument list too long */
#define  ENOEXEC     8 /* Exec format error */
#define  EBADF     9 /* Bad file number */
#define  ECHILD      10 /* No child processes */
#define  EAGAIN      11 /* Try again */
#define  ENOMEM      12 /* Out of memory */
#define  EACCES      13 /* Permission denied */
#define  EFAULT      14 /* Bad address */
#define  ENOTBLK     15 /* Block device required */
#define  EBUSY    16 /* Device or resource busy */
#define  EEXIST      17 /* File exists */
#define  EXDEV    18 /* Cross-device link */
#define  ENODEV      19 /* No such device */
#define  ENOTDIR     20 /* Not a directory */
#define  EISDIR      21 /* Is a directory */
#define  EINVAL      22 /* Invalid argument */
#define  ENFILE      23 /* File table overflow */
#define  EMFILE      24 /* Too many open files */
#define  ENOTTY      25 /* Not a typewriter */
#define  ETXTBSY     26 /* Text file busy */
#define  EFBIG    27 /* File too large */
#define  ENOSPC      28 /* No space left on device */
#define  ESPIPE      29 /* Illegal seek */
#define  EROFS    30 /* Read-only file system */
#define  EMLINK      31 /* Too many links */
#define  EPIPE    32 /* Broken pipe */
#define  EDOM     33 /* Math argument out of domain of func */
#define  ERANGE      34 /* Math result not representable */

#endif
```

Fonte: Fuse (2009).

Quadro 5 – Definições dos erros (`errno`) do FUSE

O código de um sistema de arquivos para FUSE resume-se em uma função `main` e em outras funções referentes a cada uma das operações do FUSE que o sistema de arquivos implementará.

No Quadro 6 é apresentado um exemplo de um sistema de arquivos desenvolvido para FUSE.

```c
/*
    FUSE: Filesystem in Userspace
    Copyright (C) 2001-2005  Miklos Szeredi <miklos@szeredi.hu>

    This program can be distributed under the terms of the GNU GPL.
    See the file COPYING.
*/

#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

static const char *hello_str = "Hello World!\n";
static const char *hello_path = "/hello";

static int hello_getattr(const char *path, struct stat *stbuf)
{
    int res = 0;

    memset(stbuf, 0, sizeof(struct stat));
    if(strcmp(path, "/") == 0) {
        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 2;
    }
    else if(strcmp(path, hello_path) == 0) {
        stbuf->st_mode = S_IFREG | 0444;
        stbuf->st_nlink = 1;
        stbuf->st_size = strlen(hello_str);
    }
    else
        res = -ENOENT;

    return res;
}

static int hello_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                         off_t offset, struct fuse_file_info *fi)
{
    (void) offset;
    (void) fi;

    if(strcmp(path, "/") != 0)
        return -ENOENT;

    filler(buf, ".", NULL, 0);
    filler(buf, "..", NULL, 0);
    filler(buf, hello_path + 1, NULL, 0);

    return 0;
}

static int hello_open(const char *path, struct fuse_file_info *fi)
{
    if(strcmp(path, hello_path) != 0)
        return -ENOENT;

    if((fi->flags & 3) != O_RDONLY)
        return -EACCES;

    return 0;
}

static int hello_read(const char *path, char *buf, size_t size, off_t offset,
                      struct fuse_file_info *fi)
{
    size_t len;
    (void) fi;
    if(strcmp(path, hello_path) != 0)
        return -ENOENT;

    len = strlen(hello_str);
    if (offset < len) {
        if (offset + size > len)
            size = len - offset;
        memcpy(buf, hello_str + offset, size);
```

```
    } else
        size = 0;

    return size;
}

static struct fuse_operations hello_oper = {
    .getattr       = hello_getattr,
    .readdir       = hello_readdir,
    .open = hello_open,
    .read = hello_read,
};

int main(int argc, char *argv[])
{
    return fuse_main(argc, argv, &hello_oper);
}
```

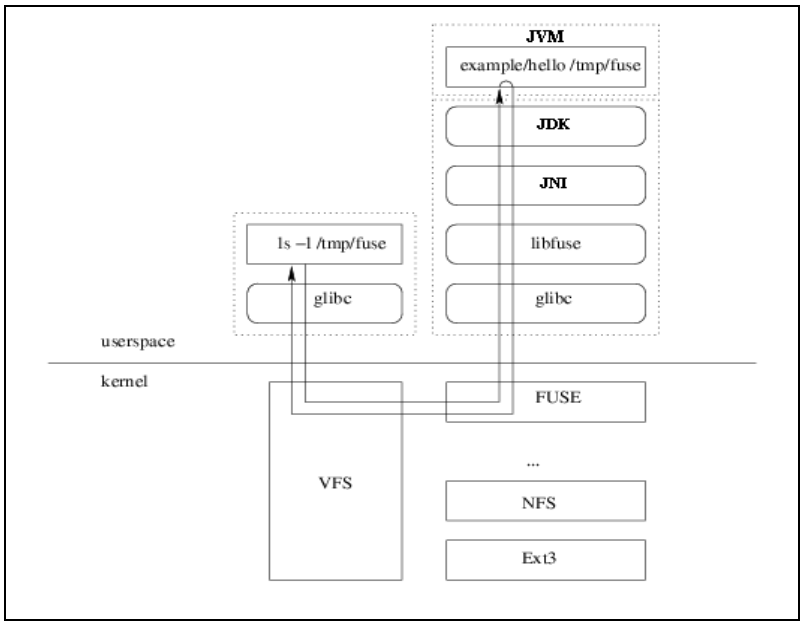
Fonte: Fuse (2009).

Quadro 6 – Exemplo de um sistema de arquivos desenvolvido para FUSE

2.3.2 FILE SYSTEM IN USER SPACE JAVA API (FUSE-J)

Segundo Fuse-j (2009), o FUSE-J é uma API Java que usa ligações JNI (*Java Native Interface*) à biblioteca FUSE e permite escrever sistemas de arquivos Linux em linguagem Java.

A diferença entre o FUSE e o FUSE-J, é que o primeiro, como já foi apresentado, é uma camada executada no espaço do *Kernel* que disponibiliza uma interface mais simples para a programação de sistemas de arquivos em C e a segunda, é uma interface Java para o desenvolvimento de sistemas de arquivos que são executados sobre FUSE. A Figura 11 apresenta o fluxo de execução de uma operação no FUSE-J.



Fonte: adaptado de Fuse (2009).

Figura 11 – Exemplo do fluxo de uma chamada de sistema utilizando o FUSE-J

O FUSE-J possui uma interface JAVA para possibilitar o desenvolvimento do sistema de arquivos nesta linguagem e fazer, conseqüente, com que o sistema de arquivos seja executado sobre a plataforma JVM, no espaço do usuário. As funções disponíveis nessa interface podem ser vistas na Figura 12.

```
public interface Filesystem1 extends FilesystemConstants
{
   public FuseStat getattr(String path) throws FuseException;

   public String readlink(String path) throws FuseException;

   public FuseDirEnt[] getdir(String path) throws FuseException;

   public void mknod(String path, int mode, int rdev) throws FuseException;

   public void mkdir(String path, int mode) throws FuseException;

   public void unlink(String path) throws FuseException;

   public void rmdir(String path) throws FuseException;

   public void symlink(String from, String to) throws FuseException;

   public void rename(String from, String to) throws FuseException;

   public void link(String from, String to) throws FuseException;

   public void chmod(String path, int mode) throws FuseException;

   public void chown(String path, int uid, int gid) throws FuseException;

   public void truncate(String path, long size) throws FuseException;

   public void utime(String path, int atime, int mtime) throws FuseException;

   public FuseStatfs statfs() throws FuseException;

   public void open(String path, int flags) throws FuseException;

   public void read(String path, ByteBuffer buf, long offset) throws FuseException;

   public void write(String path, ByteBuffer buf, long offset) throws FuseException;

   public void release(String path, int flags) throws FuseException;
}
```

Fonte: adaptado de Fuse-j (2009).

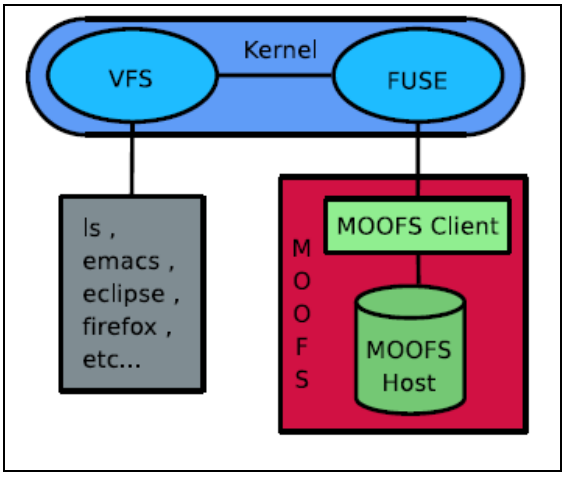
Figura 12 – Interface do FUSE-J para programação do sistema de arquivos

2.4 TRABALHOS CORRELATOS

Durante a fase de levantamento bibliográfico foram identificados três trabalhos correlatos: o *Mobile Objects Oriented File System* (MOOFS), o BloggerFS e o ClamFS.

O MOOFS é um projeto desenvolvido na linguagem Phyton, na Universisdade de São

Paulo, por Mizaki, Diez e Lundberg (2007), que tem como objetivo a criação de um sistema de arquivos distribuído em que cada arquivo encontra-se espalhado em vários computadores. Nele nenhuma máquina armazena o arquivo inteiro, apenas pequenos blocos de dados. Segundo os autores, o MOOFS é um sistema de arquivos oportunista, pois os blocos são armazenados na máquina que tiver mais recursos disponíveis, além de garantir a redundância dos dados. A Figura 13 apresenta a arquitetura do MOOFS, que se utiliza de um cliente e um servidor, ambos instalados na mesma máquina, para possibilitar a comunicação entre seus agentes.

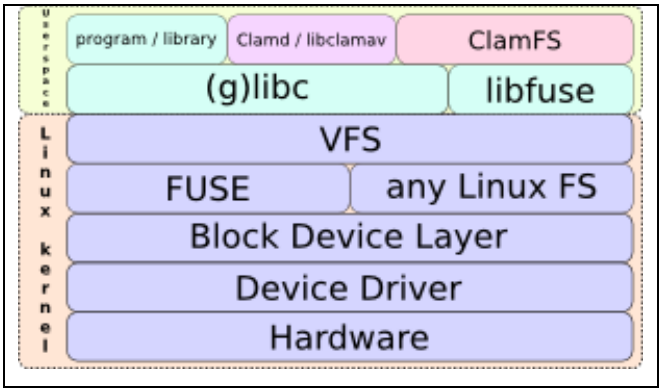


Fonte: Mizaki, Diez e Lundberd (2007).
Figura 13 – Arquitetura do MOOFS

O sistema de arquivos BloggerFS (*Blogger File System*) foi desenvolvido por Neto (2006) para que os usuários possam utilizar as funcionalidades de um sistema operacional para manipular as postagens em seus *blogs* de suas próprias máquinas. No BloggerFS Para ter acesso ao conteúdo do *blog*, o usuário só precisa informar ao sistema operacional um ponto de montagem, o usuário e a senha do *blog* em questão. Este sistema operacional, além de executar sobre o FUSE, foi desenvolvido em Java fazendo uso também do FUSE-J.

O ClamFS (Clam *File System*) desenvolvido por Burghardt (2009), é um sistema de arquivos que utiliza o antivírus ClamAV (Clam *Anti Virus*), comum no Linux, para verificar a existência de vírus em seus arquivos, em tempo de execução. Conforme os seus arquivos vão sendo manipulados, o ClamFS, através do *daemon*[2] `clamd` do ClamAV, efetua a varredura do mesmo enviando um *e-mail* ao administrador caso um vírus seja encontrado. A Figura 14 apresenta a arquitetura do ClamFS.

[2] Segundo Watson (2004), os *daemons* do Linux são processos (serviços) executados em segundo plano, que ficam aguardando requisições de outros processos.



Fonte: Burghardt (2009).

Figura 14 – Arquitetura do sistema de arquivos ClamFS

3 DESENVOLVIMENTO

Para detalhar o desenvolvimento, o presente capítulo apresenta a análise e especificação dos requisitos funcionais e não funcionais do sistema, a especificação através de diagramas de casos de uso, atividades e de classe e os detalhes sobre a implementação do protótipo.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O sistema de arquivos deverá atender os seguintes requisitos funcionais:

a) permitir a montagem do sistema de arquivos em um diretório do sistema operacional;
b) permitir que os arquivos e diretórios que forem criados ou copiados para o ponto de montagem sejam armazenados dentro de um arquivo (encapsulamento);
c) permitir que o encapsulamento gerado possa ser montado em um diretório do sistema operacional, para manipulação de seu conteúdo.

O sistema de arquivos deverá atender os seguintes requisitos não funcionais:

a) ser implementado sobre a plataforma FUSE;
b) ser implementado utilizando o ambiente NetBeans IDE (*Integrated Development Environment*) versão 6.7;
c) ser implementado na linguagem Java versão 1.5;
d) ser implementado utilizando a interface FUSE-J
e) ser compatível com as distribuições GNU/Linux.

3.2 ESPECIFICAÇÃO

Nesta seção serão apresentados a arquitetura do sistema, os diagramas de casos de uso, de atividades e de classes. Os diagramas foram desenvolvidos utilizando a ferramenta Enterprise Architect 5.0 da Sparx Systems.

3.2.1 Arquitetura do sistema

O sistema é baseado no conceito de arquivo de encapsulamento. Este arquivo é análogo a um dispositivo de armazenamento, pois armazena o resultado do processo de serialização dos objetos referentes aos arquivos, diretórios e *links* criados pelo usuário.

3.2.2 Diagrama de casos de uso

Os diagramas de casos de uso aqui apresentados tem como ator o usuário que fará a utilização do sistema de arquivos desenvolvido, com base em suas necessidades do cotidiano e o próprio sistema, que realizará as operações após receber as solicitações do FUSE.

A Figura 15 apresenta uma situação onde o usuário manipula o conteúdo do sistema de arquivos, montando-o em um diretório do sistema operacional, acessando o mesmo através do ponto de montagem, executando comandos dentro do sistema de arquivos montado e desmontando-o no final da manipulação.

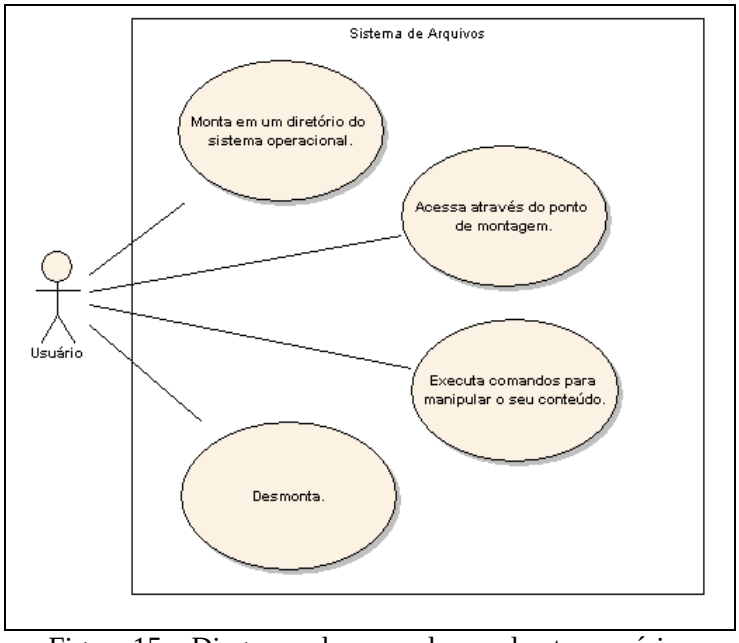


Figura 15 – Diagrama de casos de uso do ator usuário

Na Figura 16 é apresentada uma situação onde o sistema de arquivos carrega o conteúdo de seu arquivo de encapsulamento (caso existir) para a memória, ao ser montado, realiza as operações necessárias para manipular seu conteúdo (arquivos e diretórios) na memória e grava o seu conteúdo no arquivo de encapsulamento ao ser desmontado, gerando um novo, caso ainda não exista, ou atualizando o existente.

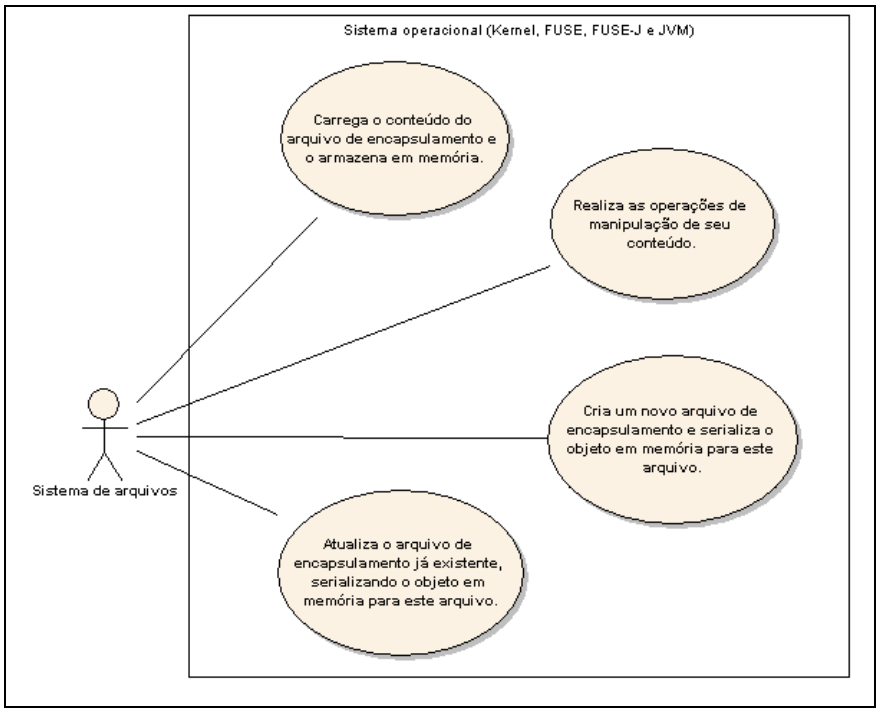


Figura 16 – Diagrama de casos de uso do ator sistema de arquivos

3.2.3 Diagrama de atividades

O diagrama de atividades apresentado na Figura 17 tem como objetivo demonstrar o procedimento de inicialização do sistema de arquivos, após o usuário solicitar o seu mapeamento em algum diretório do sistema operacional.

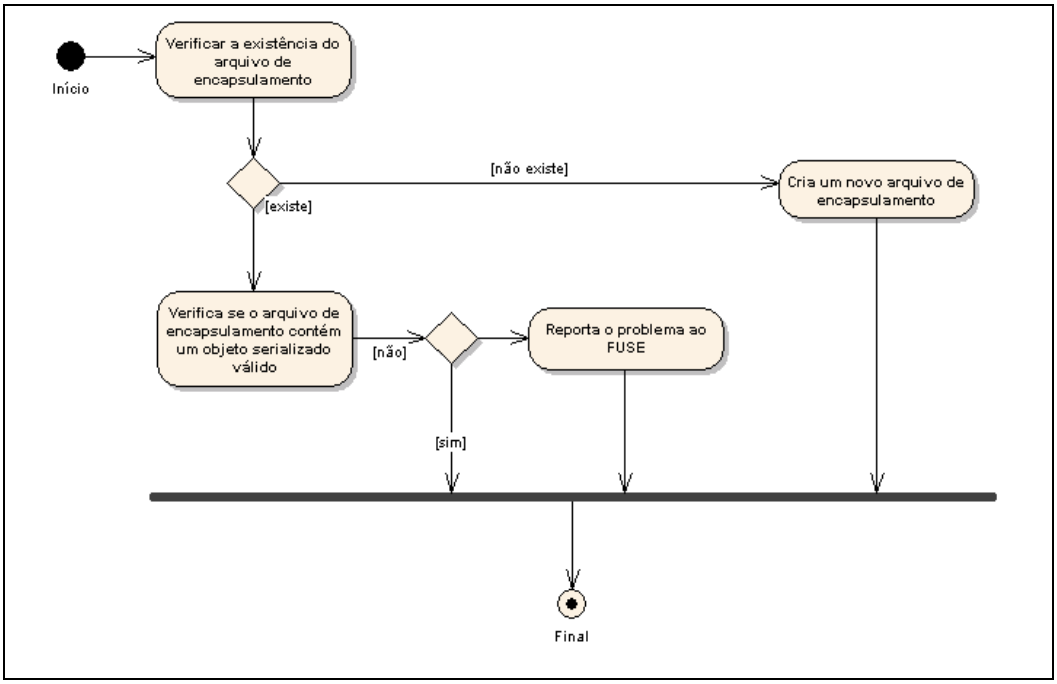


Figura 17 – Diagrama de atividades do procedimento de criação do arquivo de encapsulamento

3.2.4 Diagrama de classes

O diagrama de classes aqui apresentado, representa a estrutura de dados utilizada na concepção do sistema de arquivos denominado TKFFS. As classes são identificadas com o prefixo TKF, conforme a Figura 18.

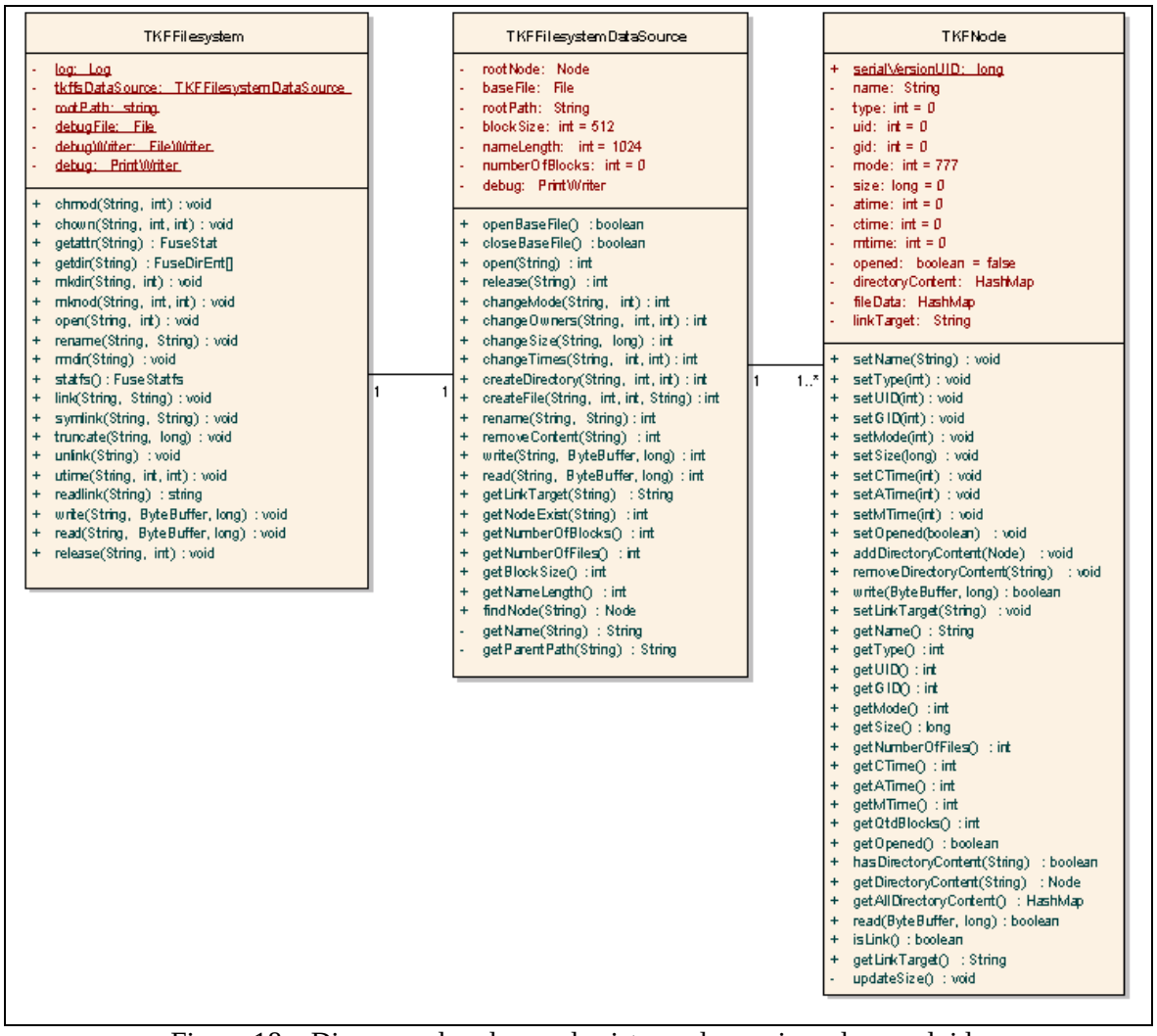


Figura 18 – Diagrama das classes do sistema de arquivos desenvolvido

A seguir cada uma das classes é apresentada em detalhes.

3.3 IMPLEMENTAÇÃO

A seguir é detalhada a implementação das classes e é mostrada a operacionalidade da implementação.

3.3.1 Classes

Para a implementação das classes, foi utilizada a ferramenta NetBeans IDE 6.7, com a plataforma JDK (Java *Development Kit*) 1.5.

3.3.1.1 Classe `TKFFilesystem`

A classe `TKFFilesystem` é utilizada como a classe de acesso ao sistema de arquivos. É esta classe que recebe as requisições do FUSE e implementa os métodos da interface FUSE-J. Além disso, ela realiza os procedimentos necessários no momento da montagem do sistema de arquivos em algum diretório do sistema operacional e de sua desmontagem.

Esta classe possui os seguintes atributos:

a) `log`: responsável por armazenar informações sobre a execução, em um arquivo de *log*;
b) `tkffsDataSource`: objeto da classe `TKFFilesystemDataSource`, que possibilita a interação entre a classe `TKFFilesystem` e o conteúdo do sistema de arquivos;
c) `rootPath`: armazena o caminho absoluto do diretório raiz do sistema de arquivos desenvolvido. Seu valor padrão é: `/`.

Os métodos desta classe são implementações da interface FUSE-J e eles, em geral, realizam chamadas para métodos da classe `TKFFilesystemDataSource`, disparando exceções para o FUSE, contendo o identificador `errno`, em caso de erros. As descrições destes métodos são apresentadas no Quadro 7.

Dentre os métodos citados, em seguida são detalhados os mais importantes, do ponto de vista da compreensão do funcionamento do sistema.

MÉTODO	DESCRIÇÃO
chmod	Altera as permissões de um determinado *node*.
chown	Altera o dono e o grupo de um determinado *node*.
getattr	Encaminha ao FUSE os atributos de um determinado *node*, como usuário, grupo, permissões, tamanho, entre outros.
getdir	Encaminha ao FUSE a lista do conteúdo de um determinado diretório.
mkdir	Cria um novo diretório.
mknod	Cria um novo arquivo.
open	Define determinado *node* como aberto.
rename	Altera o nome ou o caminho completo de um determinado *node*.
rmdir	Remove determinado diretório.
statfs	Encaminha ao FUSE informações sobre o sistema operacional.
link	Cria um *link* para determinado arquivo ou diretório.
symlink	Encaminha ao FUSE uma exceção informando que a criação de um *link* não é uma operação permitida no TKFFS.
trucate	Altera o tamanho de determinado arquivo.
unlink	Remove determinado arquivo ou *link*.
utime	Altera os tempos de criação (`ctime`), último acesso (`atime`) e última modificação (`mtime`) de um determinado *node*.
readlink	Encaminha ao FUSE o caminho absoluto ao alvo de um determinado *link*.
write	Joga o conteúdo do *buffer* recebido do FUSE em uma determinada posição de um determinado arquivo.
read	Encaminha ao FUSE um *buffer* contendo determinado conteúdo de um determinado arquivo.
release	Libera um *node* que estava definido como aberto.
main	Efetua os procedimentos necessários para inicialização (montagem) e encerramento (desmontagem) do TKFFS.

Quadro 7 – Métodos da classe `TKFFilesystem`

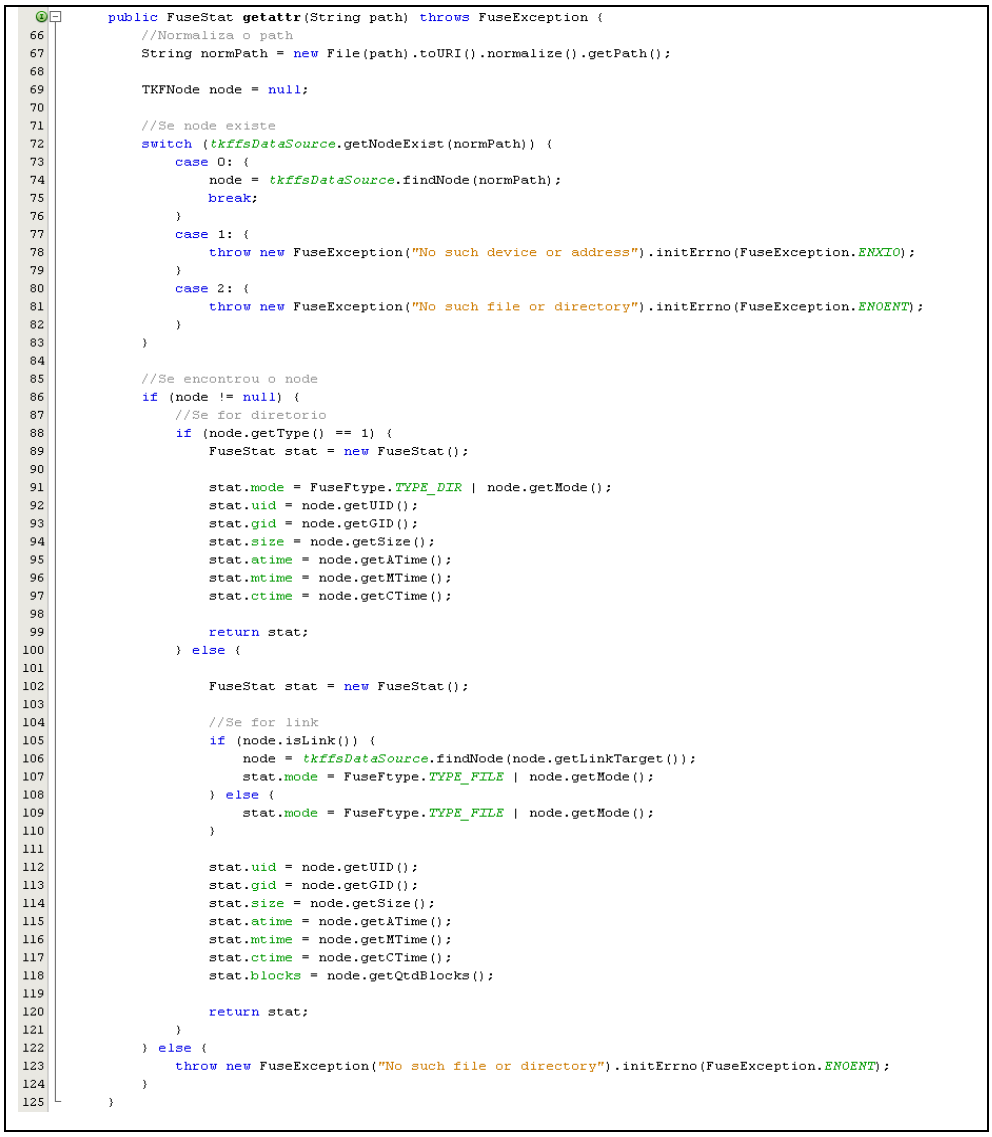
3.3.1.1.1 Método `getattr`

A função deste método é encaminhar ao FUSE os atributos de um determinado *node*, como dono, grupo, permissões, tamanho, entre outros que normalmente são armazenados na estrutura `inode` dos sistemas de arquivos Linux. No caso deste trabalho, tais atributos pertencem à classe `TKFNode`.

Entre as linhas 72 e 83, presentes na Figura 19, é tratado o retorno da chamada efetuada para o método `getNodeExist` da classe `TKFFilesystemDataSource`, passando como parâmetro o caminho absoluto do *node* em questão, a fim de verificar se ele existe. A interpretação desses retornos é feita da seguinte forma:

a) 0: o *node* procurado existe. Neste caso ele é armazenado no atributo local `node`;
b) 1: dispositivo ou endereço inexistente. Uma exceção é disparada ao FUSE contendo o identificador do problema (`errno`);
c) 2: arquivo ou diretório inexistente. Uma exceção é disparada ao FUSE contendo o identificador do problema (`errno`).

As linhas 91 a 97 são referentes ao carregamento dos atributos caso o *node* em questão seja um diretório. O código entre linhas 105 a 110 é responsável por obter o *node* alvo, caso o atual seja um *link*. Já nas linhas 112 a 118 é tratado o carregamento dos atributos, caso o *node* em questão seja um arquivo.



```
66      public FuseStat getattr(String path) throws FuseException {
67          //Normaliza o path
68          String normPath = new File(path).toURI().normalize().getPath();
69
70          TKFNode node = null;
71
72          //Se node existe
73          switch (tkffsDataSource.getNodeExist(normPath)) {
74              case 0: {
75                  node = tkffsDataSource.findNode(normPath);
76                  break;
77              }
78              case 1: {
79                  throw new FuseException("No such device or address").initErrno(FuseException.ENXIO);
80              }
81              case 2: {
82                  throw new FuseException("No such file or directory").initErrno(FuseException.ENOENT);
83              }
84          }
85
86          //Se encontrou o node
87          if (node != null) {
88              //Se for diretorio
89              if (node.getType() == 1) {
90                  FuseStat stat = new FuseStat();
91
92                  stat.mode = FuseFtype.TYPE_DIR | node.getMode();
93                  stat.uid = node.getUID();
94                  stat.gid = node.getGID();
95                  stat.size = node.getSize();
96                  stat.atime = node.getATime();
97                  stat.mtime = node.getMTime();
98                  stat.ctime = node.getCTime();
99
100                 return stat;
101             } else {
102
103                 FuseStat stat = new FuseStat();
104
105                 //Se for link
106                 if (node.isLink()) {
107                     node = tkffsDataSource.findNode(node.getLinkTarget());
108                     stat.mode = FuseFtype.TYPE_FILE | node.getMode();
109                 } else {
110                     stat.mode = FuseFtype.TYPE_FILE | node.getMode();
111                 }
112
113                 stat.uid = node.getUID();
114                 stat.gid = node.getGID();
115                 stat.size = node.getSize();
116                 stat.atime = node.getATime();
117                 stat.mtime = node.getMTime();
118                 stat.ctime = node.getCTime();
119                 stat.blocks = node.getQtdBlocks();
120
121                 return stat;
122             }
123         } else {
124             throw new FuseException("No such file or directory").initErrno(FuseException.ENOENT);
125         }
    }
```

Figura 19 – Método `getattr`

3.3.1.1.2 Método `getdir`

A função deste método é encaminhar ao FUSE um vetor contendo o conteúdo de um determinado diretório.

Entre as linhas 134 e 145, presentes na Figura 20, é tratado o retorno da chamada efetuada para o método `getNodeExist` da classe `TKFFilesystemDataSource`, passando

como parâmetro o caminho absoluto do *node* que representa o diretório do qual se quer o conteúdo, a fim de verificar se ele existe. A interpretação desses retornos é feita da seguinte forma:

a) 0: o *node* procurado existe. Neste caso ele é armazenado no atributo local `node`;
b) 1: dispositivo ou endereço inexistente. Uma exceção é disparada ao FUSE contendo o identificador do problema (`errno`);
c) 2: arquivo ou diretório inexistente. Uma exceção é disparada ao FUSE contendo o identificador do problema (`errno`).

Após isso, na linha 150 é verificado se o *node* em questão é mesmo um diretório, caso não seja, uma exceção é disparada. Se tratar-se de um diretório, o código entre as linhas 152 e 182 trata de colocar o conteúdo do diretório em um vetor de objetos específicos e encaminhá-lo ao FUSE.

```
128     public FuseDirEnt[] getdir(String path) throws FuseException {
129         //Normaliza o path
130         String normPath = new File(path).toURI().normalize().getPath();
131 
132         TKFNode node = null;
133 
134         //Se node existe
135         switch (tkffsDataSource.getNodeExist(normPath)) {
136             case 0: {
137                 node = tkffsDataSource.findNode(normPath);
138                 break;
139             }
140             case 1: {
141                 throw new FuseException("No such device or address").initErrno(FuseException.ENXIO);
142             }
143             case 2: {
144                 throw new FuseException("No such file or directory").initErrno(FuseException.ENOENT);
145             }
146         }
147 
148         //Se encontrou o node
149         if (node != null) {
150             //Se for diretorio
151             if (node.getType() == 1) {
152 
153                 HashMap directoryContent = node.getAllDirectoryContent();
154                 Collection col = directoryContent.values();
155                 Iterator nodesIterator = col.iterator();
156 
157                 FuseDirEnt[] dirEnts = new FuseDirEnt[directoryContent.size()];
158 
159                 int index = 0;
160 
161                 while (nodesIterator.hasNext()) {
162                     FuseDirEnt dirEnt = new FuseDirEnt();
163                     dirEnts[index] = dirEnt;
164                     node = (TKFNode) nodesIterator.next();
165 
166                     //Se for diretorio
167                     if (node.getType() == 1) {
168                         dirEnt.mode = FuseFtype.TYPE_DIR | node.getMode();
169                     } else {
170                         //Se for link
171                         if (node.isLink()) {
172                             dirEnt.mode = FuseFtype.TYPE_SYMLINK | node.getMode();
173                         } else {
174                             dirEnt.mode = FuseFtype.TYPE_FILE | node.getMode();
175                         }
176                     }
177 
178                     dirEnt.name = node.getName();
179 
180                     index++;
181                 }
182 
183                 return dirEnts;
184             } else {
185                 throw new FuseException("Not a directory: " + path).initErrno(FuseException.ENOTDIR);
186             }
187         } else {
188             throw new FuseException("No such file or directory").initErrno(FuseException.ENOENT);
189         }
    }
```

Figura 20 – Método `getdir`

3.3.1.1.3 Método `statfs`

A função deste método é encaminhar ao FUSE os atributos do sistema de arquivos, são eles: quantidade de blocos, quantidade de arquivos, tamanho do bloco e tamanho máximo dos nomes. Atributos como estes são encontrados normalmente na estrutura `superblock` dos sistemas de arquivos Linux. No caso deste trabalho, tais atributos pertencem à classe `TKFFilesystemDataSource`.

Entre as linhas 299 e 306, presentes na Figura 21, os atributos são adicionados a um objeto específico e encaminhados ao FUSE.

```
    public FuseStatfs statfs() throws FuseException {
299     FuseStatfs statfs = new FuseStatfs();
300
301     statfs.blocks = tkffsDataSource.getNumberOfBlocks();
302     statfs.files = tkffsDataSource.getNumberOfFiles();
303     statfs.blockSize = tkffsDataSource.getBlockSize();
304     statfs.namelen = tkffsDataSource.getNameLength();
305
306     return statfs;
307 }
```

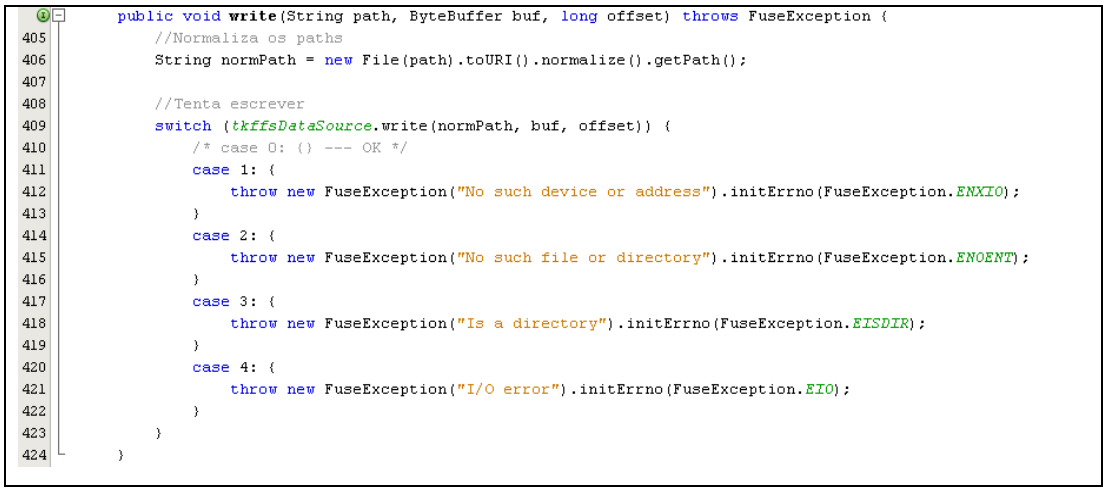
Figura 21 – Método `statfs`

3.3.1.1.4 Método `write`

A função deste método é jogar o conteúdo do *buffer* recebido do FUSE em uma determinada posição de um determinado arquivo.

Na linha 409, presente na Figura 22, é tratado o retorno da chamada feita para o método `write` da classe `TKFFilesystemDataSource`, passando como parâmetro o caminho absoluto do arquivo em questão, o *buffer* encaminhado pelo FUSE e a posição onde o conteúdo do *buffer* será escrito no arquivo. A interpretação desses retornos é feita da seguinte forma:

a) 0: a escrita ocorreu com sucesso;
b) 1: dispositivo ou endereço inexistente. Uma exceção é disparada ao FUSE contendo o identificador do problema (`errno`);
c) 2: arquivo ou diretório inexistente. Uma exceção é disparada ao FUSE contendo o identificador do problema (`errno`);
d) 3: o caminho em questão é referente a um diretório;
e) 4: erro de entrada e saída.



```
405     public void write(String path, ByteBuffer buf, long offset) throws FuseException {
406         //Normaliza os paths
407         String normPath = new File(path).toURI().normalize().getPath();
408 
409         //Tenta escrever
410         switch (tkffsDataSource.write(normPath, buf, offset)) {
411             /* case 0: () --- OK */
412             case 1: {
413                 throw new FuseException("No such device or address").initErrno(FuseException.ENXIO);
414             }
415             case 2: {
416                 throw new FuseException("No such file or directory").initErrno(FuseException.ENOENT);
417             }
418             case 3: {
419                 throw new FuseException("Is a directory").initErrno(FuseException.EISDIR);
420             }
421             case 4: {
422                 throw new FuseException("I/O error").initErrno(FuseException.EIO);
423             }
424         }
    }
```

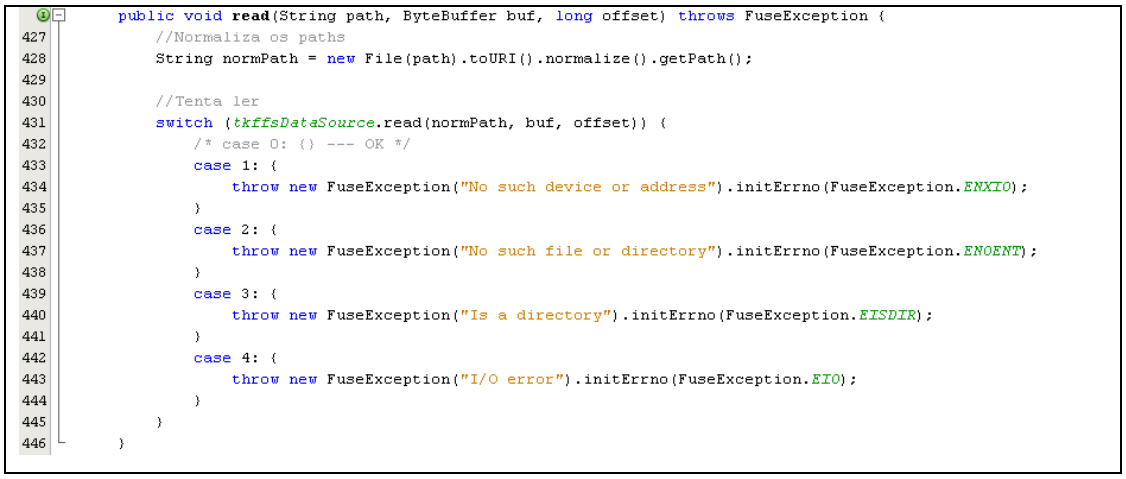
Figura 22 – Método `write`

3.3.1.1.5 Método `read`

A função deste método é encaminhar ao FUSE um *buffer* contendo um determinado conteúdo de um determinado arquivo.

Na linha 431, presente na Figura 23, é tratado o retorno da chamada feita para o método `read` da classe `TKFFilesystemDataSource`, passando como parâmetro o caminho absoluto do arquivo em questão, o *buffer* encaminhado pelo FUSE e a posição onde está o conteúdo do arquivo que será copiado para o *buffer*. A interpretação desses retornos é feita da seguinte forma:

a) 0: a leitura ocorreu com sucesso;
b) 1: dispositivo ou endereço inexistente. Uma exceção é disparada ao FUSE contendo o identificador do problema (`errno`);
c) 2: arquivo ou diretório inexistente. Uma exceção é disparada ao FUSE contendo o identificador do problema (`errno`);
d) 3: o caminho em questão é referente a um diretório;
e) 4: erro de entrada e saída.



```
427     public void read(String path, ByteBuffer buf, long offset) throws FuseException {
428         //Normaliza os paths
429         String normPath = new File(path).toURI().normalize().getPath();
430 
431         //Tenta ler
432         switch (tkffsDataSource.read(normPath, buf, offset)) {
433             /* case 0: {} --- OK */
434             case 1: {
435                 throw new FuseException("No such device or address").initErrno(FuseException.ENXIO);
436             }
437             case 2: {
438                 throw new FuseException("No such file or directory").initErrno(FuseException.ENOENT);
439             }
440             case 3: {
441                 throw new FuseException("Is a directory").initErrno(FuseException.EISDIR);
442             }
443             case 4: {
444                 throw new FuseException("I/O error").initErrno(FuseException.EIO);
445             }
446         }
    }
```

Figura 23 – Método `read`

3.3.1.1.6 Método `main`

Este método é o responsável pela inicialização do TKFFS, quando o mesmo for montado em um diretório do sistema e pelo seu encerramento, quando desmontado.

Na linha 475, presente na Figura 24, é criado um objeto da classe `TKFFilesystemDataSource`, que serve como ponto de acessos ao conteúdo do sistema de arquivos, passando como parâmetro o caminho absoluto do seu diretório raiz (`/`) e o objeto referente ao arquivo de encapsulamento.

Na linha 478 é que o TKFFS é verdadeiramente inicializado. Chamando o método `openBaseFile` da classe `TKFFilesystemDataSource`, o arquivo de encapsulamento é carregado, ficando disponível para manipulação. Em caso de erro, uma exceção é disparada ao FUSE informando que o arquivo de encapsulamento não pôde ser lido.

Já na linha 482, é solicitado ao FUSE a montagem, melhor dizendo, a disponibilização do acesso ao conteúdo do arquivo de encapsulamento, através de ponto de montagem.

Entre as linhas 487 e 491, é realizado o procedimento de finalização do TKFFS, após o FUSE efetuar sua desmontagem. Este procedimento realiza a gravação do conteúdo do sistema de arquivos que estava na memória, no arquivo de encapsulamento e a liberação do mesmo.

```
464      public static void main(String[] args) {
465          try {
466              if (args.length < 1) {
467                  System.exit(-1);
468              }
469
470              String fuseArgs[] = new String[args.length - 1];
471              System.arraycopy(args, 0, fuseArgs, 0, fuseArgs.length);
472              String pathTKFFile = new File(args[args.length - 1]).toURI().normalize().getPath();
473              File tkfFile = new File(pathTKFFile);
474
475              tkffsDataSource = new TKFFilesystemDataSource(rootPath, tkfFile);
476
477              //Se conseguir ler o baseFile
478              if (tkffsDataSource.openBaseFile()) {
479                  log.info("entering");
480
481                  try {
482                      FuseMount.mount(fuseArgs, new TKFFilesystem());
483                  } catch (Exception e) {
484                      log.error(e.getMessage());
485                  } finally {
486                      //Se nao conseguir fechar o basFile
487                      if (tkffsDataSource.closeBaseFile()) {
488                          log.info("exiting");
489                      } else {
490                          log.error("impossible to write the base file");
491                      }
492                  }
493              } else {
494                  log.error("impossible to read the base file");
495              }
496          } catch (Exception e) {
497          }
498      }
499  }
```

Figura 24 – Método `main`

3.3.1.2 Classe `TKFFilesystemDataSource`

A classe `TKFFilesystemDataSource` é utilizada para gerenciar os dados que são armazenados no arquivo de encapsulamento, a partir de chamadas realizadas pela classe `TKFFilesystem`.

Esta classe possui os seguintes atributos:

a) `rootNode`: quando vemos a estrutura de arquivos como uma árvore de diretórios, este atributo representa o nó raiz, abaixo do qual estão todos os outros arquivos, diretórios e *links*;

b) `baseFile`: é o arquivo de encapsulamento, análogo a um dispositivo de armazenamento. É neste arquivo que os dados serão literalmente gravados;

c) `rootPath`: armazena o caminho absoluto (**/**) do diretório raiz do sistema de arquivos desenvolvido;

d) `blockSize`: representa o tamanho dos blocos do sistema de arquivos. Como o TKFFS não trabalha com blocos e sim com serialização de objetos, este dado é

armazenado em um padrão estático igual a **4** ***Kilobytes***. Esta informação é necessária apenas por ser solicitada pelo FUSE;

e) `nameLength`: representa o tamanho máximo dos nomes dos arquivos, diretórios e *links*. Este dado é armazenado em um padrão estático igual a **255**;

f) `numberOfBlocks`: representa a quantidade de blocos virtuais do sistema de arquivos. Seu valor inicial é igual a **0**.

Os métodos desta classe são responsáveis pela manipulação do conteúdo do sistema de arquivos. Em sua grande maioria são chamados por métodos da classe `TKFFilesystem`, para realizar as operações solicitadas pelo usuário através de sua comunicação com o VFS e consecutivamente com o FUSE. As descrições dos métodos desta classe são apresentadas no Quadro 8.

MÉTODO	DESCRIÇÃO
openBaseFile	Verifica se o arquivo de encapsulamento possui um objeto serializado válido de `TKFNode`, para então carregá-lo para a memória. Este objeto representa o *node* raiz do sistema de arquivos e todo o seu conteúdo.
closeBaseFile	Serializa o objeto referente ao *node* raiz do sistema de arquivos e todo o seu conteúdo, para poder armazená-lo no arquivo de encapsulamento.
open	Sinaliza determinado arquivo como aberto.
release	Sinaliza determinado arquivo como liberado.
changeMode	Modifica a permissão de determinado *node*.
changeOwners	Modifica o proprietário e o grupo de determinado *node*.
changeSize	Modifica o tamanho de determinado arquivo.
changeTimes	Modifica os tempos de criação (`ctime`), último acesso (`atime`) e última modificação (`mtime`) de um determinado *node*.
createDirectory	Cria um novo diretório.
createFile	Cria um novo arquivo ou *link*.
rename	Modifica o nome ou o caminho completo de um determinado *node*.
removeContent	Remove um determinado conteúdo de um determinado diretório.
write	Joga o conteúdo do *buffer* recebido do FUSE em uma determinada posição de um determinado arquivo.
read	Encaminha ao FUSE um *buffer* contendo determinado conteúdo de um determinado arquivo.
getLinkTarget	Retorna o caminho absoluto do alvo de um determinado *link*.
getNodeExist	Verifica se um determinado *node* existe.
getNumberOfBlocks	Calcula e retorna o número de blocos do sistema de arquivos, com base na informação obtida pelo método `getQtdBlocks`, da classe `TKFNode`.
getNumberOfFiles	Retorna o número de arquivos do sistema de arquivos.
getBlockSize	Retorna o tamanho dos blocos virtuais do sistema de arquivos.
getNameLength	Retorna o tamanho máximo para o nome de arquivos e diretórios.
findNode	Procura por um determinado *node*.
getName	Retorna apenas o nome do *node*, com base em seu caminho absoluto.
getParentPath	Retorna apenas o caminho do diretório onde determinado *node* se encontra, com base em seu caminho absoluto.

Quadro 8 – Métodos da classe `TKFFilesystemDataSource`

Dentre os métodos citados, em seguida são detalhados os mais importantes, do ponto de vista da compreensão do funcionamento do sistema.

3.3.1.2.1 Método `openBaseFile`

A função deste método é carregar o conteúdo do arquivo de encapsulamento para a memória ou criar um novo, caso ainda não exista.

Na linha 49, presente na Figura 25, é verificado se o arquivo de encapsulamento existe, caso não existir, o método `closeBaseFile` é chamado, para que o arquivo seja criado.

Após garantir a existência de um arquivo de encapsulamento, a linha 64 verifica se tal arquivo possui um objeto serializado válido da classe `TKFNode`, referente ao *node* raiz do sistema de arquivos e todo o seu conteúdo. Caso contiver um objeto válido, ele será carregado para a memória, sendo armazenado na variável `rootNode` (linha 66).

```
45      /* Open Base File */
46      public boolean openBaseFile() {
47          try {
48              //Se o baseFile ainda nao existir
49              if (!baseFile.exists()) {
50                  //Cria
51                  closeBaseFile();
52              }
53
54              //Carrega o aruivo usando FileInputStream
55              FileInputStream f_in = new FileInputStream(baseFile);
56
57              //Carrega o objeto usando ObjectInputStream
58              ObjectInputStream obj_in = new ObjectInputStream(f_in);
59
60              //Carrega o objeto
61              Object obj = obj_in.readObject();
62
63              //Se o objeto for um TKFNode
64              if (obj instanceof TKFNode) {
65                  //Faz um Cast do objeto para TKFNode
66                  rootNode = (TKFNode) obj;
67              } else {
68                  //Retorno: nao e um objeto valido
69                  return false;
70              }
71
72              obj_in.close();
73              f_in.close();
74          } catch (Exception e) {
75              e.printStackTrace();
76
77              //Retorno: nao conseguiu ler
78              return false;
79          }
80
81          //Retorno: conseguiu ler
82          return true;
83      }
```

Figura 25 – Método `openBaseFile`

3.3.1.2.2 Método `closeBaseFile`

A função deste método é serializar o conteúdo da variável `rootNode`, referente ao *node* raiz do sistema de arquivos e todo o seu conteúdo, para então gravá-lo no arquivo de encapsulamento.

A gravação do objeto serializado é realizado na linha 95, presente na Figura 26.

```
85      /* Close Base File */
86      public boolean closeBaseFile() {
87          try {
88              //Escreve o arquivo com FileOutputStream
89              FileOutputStream f_out = new FileOutputStream(baseFile);
90
91              //Escreve o objeto com ObjectOutputStream
92              ObjectOutputStream obj_out = new ObjectOutputStream(f_out);
93
94              //Escreve o objeto no arquivo
95              obj_out.writeObject(rootNode);
96
97              obj_out.flush();
98              obj_out.close();
99              f_out.close();
100         } catch (Exception e) {
101             e.printStackTrace();
102
103             //Retorno: nao conseguiu escrever
104             return false;
105         }
106
107         //Retorno: conseguiu escrever
108         return true;
109     }
```

Figura 26 – Método `closeBaseFile`

3.3.1.2.3 Método `crateDirectory`

A função deste método é tentar criar o diretório solicitado.

Na linha 256, presente na Figura 27, a função `getParentPath` é chamada para obter o caminho onde o diretório a ser criado será armazenado. Logo após, na linha 258, a função `findNode` é chamada para obter o *node* referente ao diretório que receberá o que será criado. Caso ele seja obtido com sucesso, na linha 263 é verificado se o mesmo já contém o diretório a ser criado, se não contiver, um novo objeto de `TKFNode` é criado e recebe seus atributos, antes de ser adicionado ao conteúdo do *node* pai (linhas 266 a 276).

Os procedimentos de criação de um arquivo ou *link* pelo método `crateFile` são idênticos aos deste método.

```
251         /* mkdir */
252         public int createDirectory(String path, int mode, int time) {
253             //Se o path comecar com /
254             if (path.charAt(0) == '/') {
255 
256                 String parentPath = getParentPath(path);
257 
258                 TKFNode nodeFinded = findNode(parentPath);
259 
260                 //Se encontrou o node
261                 if (nodeFinded != null) {
262                     //Se o node pai ainda nao tiver o diretorio
263                     if (!nodeFinded.hasDirectoryContent(getName(path))) {
264 
265                         //Cria o node filho
266                         TKFNode nodeNew = new TKFNode(1);
267 
268                         //Muda as configuracoes
269                         nodeNew.setName(getName(path));
270                         nodeNew.setMode(mode);
271                         nodeNew.setCTime(time);
272                         nodeNew.setATime(time);
273                         nodeNew.setMTime(time);
274 
275                         //Adiciona no node pai
276                         nodeFinded.addDirectoryContent(nodeNew);
277 
278                     } else {
279                         //Retorno: node ja existe
280                         return 4;
281                     }
282                 } else {
283                     //Retorno: arquivo ou diretorio nao existe
284                     return 2;
285                 }
286             } else {
287                 //Retorno: nao e um caminho valido
288                 return 1;
289             }
290 
291             return 0;
292         }
```

Figura 27 – Método `createDirectory`

3.3.1.2.4 Método `rename`

A função deste método é modificar o nome de um *node* ou alterar a sua localização na árvore de diretórios do sistema de arquivos, caso necessário.

Nas linhas 354 e 355, presentes na Figura 28, são obtidos o *node* pai atual e o *node* pai de destino, respectivamente.

Se o *node* pai atual for encontrado (linha 359) e possuir em seu conteúdo, aquele que se quer manipular (linha 362), na linha 365 tal *node* filho é obtido e pode ter seu nome alterado ou não, antes de ser armazenado de volta no *node* pai atual, ou realocado. Caso este *node* filho deva ser realocado (linha 377), ele será armazenado no *node* pai de destino (linha 381) e removido do *node* pai atual (linha 384).

```
346         /* rename */
347         public int rename(String pathFrom, String pathTo) {
348             //Se o path comecar com /
349             if (pathFrom.charAt(0) == '/' && pathTo.charAt(0) == '/') {
350 
351                 String parentPathFrom = getParentPath(pathFrom);
352                 String parentPathTo = getParentPath(pathTo);
353 
354                 TKFNode nodePaiAtual = findNode(parentPathFrom);
355                 TKFNode nodePaiNovo = findNode(parentPathTo);
356                 TKFNode nodeToRename = null;
357 
358                 //Se encontrou o node pai atual
359                 if (nodePaiAtual != null) {
360 
361                     //Se existe algo com o mesmo nome dentro do node pai atual
362                     if (nodePaiAtual.hasDirectoryContent(getName(pathFrom))) {
363 
364                         //Armazena o node
365                         nodeToRename = nodePaiAtual.getDirectoryContent(getName(pathFrom));
366 
367                         //Renomeia
368                         nodeToRename.setName(getName(pathTo));
369 
370                         //Procura o novo node pai
371                         nodePaiNovo = findNode(parentPathTo);
372 
373                         //Se encontrou o novo node pai
374                         if (nodePaiNovo != null) {
375 
376                             //Se nao existe algo com o mesmo nome dentro do novo node pai
377                             if (!nodePaiNovo.hasDirectoryContent(getName(pathTo))) {
378                                 //debug.println("DataSource: rename: O novo node pai ainda nao possui o arquivo");
379 
380                                 //Remove o node do node pai atual
381                                 nodePaiAtual.removeDirectoryContent(getName(pathFrom));
382 
383                                 //Adiciona o node no novo node pai
384                                 nodePaiNovo.addDirectoryContent(nodeToRename);
385 
386                             } else {
387                                 //Retorno: node ja existe
388                                 return 3;
389                             }
390                         } else {
391                             //Retorno: node ja existe
392                             return 2;
393                         }
394                     } else {
395                         //Retorno: node nao existe
396                         return 2;
397                     }
398                 } else {
399                     //Retorno: arquivo ou diretorio nao existe
400                     return 2;
401                 }
402             } else {
403                 //Retorno: nao e um caminho valido
404                 return 1;
405             }
406 
407             return 0;
408         }
```

Figura 28 – Método `rename`

3.3.1.2.5 Método `write`

A função deste método é tentar modificar o conteúdo de um arquivo.

As linhas 448 e 451, presentes na Figura 29, são responsáveis por procurar o *node* referente ao arquivo a ser alterado, se ele for encontrado, na linha 454 é verificado se trata-se mesmo de um arquivo. Se o *node* em questão for um arquivo, o método `write` da classe `TKFNode` é chamado para realizar a alteração dentro do objeto (linha 457), caso não for um arquivo, a linha 462 verifica se trata-se de um *link*. Se o *node* em questão for um *link*, o

presente método é chamado novamente, recebendo como parâmetro o alvo do *link* em questão.

```
443         /* write */
444         public int write(String path, ByteBuffer buf, long offset) {
445             //Se o path comecar com /
446             if (path.charAt(0) == '/') {
447 
448                 TKFNode nodeFinded = findNode(path);
449 
450                 //Se encontrou o node
451                 if (nodeFinded != null) {
452 
453                     //Se o node for um arquivo
454                     if (nodeFinded.getType() == 0) {
455 
456                         //Se nao conseguir escrever o conteudo
457                         if (!nodeFinded.write(buf, offset)) {
458                             return 4;
459                         }
460                     } else{
461                         //Se o node for um link
462                         if (nodeFinded.getType() == 2) {
463 
464                             //Tenta escrever o conteudo
465                             return write(nodeFinded.getLinkTarget(), buf, offset);
466                         } else {
467                             //Retorno: e um diretorio
468                             return 3;
469                         }
470                     }
471                 } else {
472                     //Retorno: arquivo ou diretorio nao existe
473                     return 2;
474                 }
475             } else {
476                 //Retorno: nao e um caminho valido
477                 return 1;
478             }
479 
480             return 0;
481         }
```

Figura 29 – Método `write`

3.3.1.2.6 Método `read`

A função deste método é tentar ler o conteúdo de um arquivo.

As linhas 488 e 491, presentes na Figura 30, são responsáveis por procurar o *node* referente ao arquivo a ser lido, se ele for encontrado, na linha 494 é verificado se trata-se mesmo de um arquivo. Se o *node* em questão for um arquivo, o método `read` da classe `TKFNode` é chamado para realizar a alteração dentro do objeto (linha 497), caso não for um arquivo, a linha 503 verifica se trata-se de um *link*. Se o *node* em questão for um *link*, o presente método é chamado novamente, recebendo como parâmetro o alvo do *link* em questão.

```
483         /* read */
484         public int read(String path, ByteBuffer buf, long offset) {
485             //Se o path comecar com /
486             if (path.charAt(0) == '/') {
487 
488                 TKFNode nodeFinded = findNode(path);
489 
490                 //Se encontrou o node
491                 if (nodeFinded != null) {
492 
493                     //Se o node for um arquivo
494                     if (nodeFinded.getType() == 0) {
495 
496                         //Se nao conseguir ler o conteudo
497                         if (!nodeFinded.read(buf, offset)) {
498                             //Retorno: erro de I/O
499                             return 4;
500                         }
501                     }else{
502                         //Se o node for um link
503                         if (nodeFinded.getType() == 2) {
504                             //Tenta ler o conteudo
505                             return read(nodeFinded.getLinkTarget(), buf, offset);
506                         } else {
507                             //Retorno: e um diretorio
508                             return 3;
509                         }
510                     }
511                 } else {
512                     //Retorno: arquivo ou diretorio nao existe
513                     return 2;
514                 }
515             } else {
516                 //Retorno: nao e um caminho valido
517                 return 1;
518             }
519 
520             return 0;
521         }
```

Figura 30 – Método `read`

3.3.1.3 Classe `TKFNode`

Os objetos da classe `TKFNode` representam os arquivos, *links* ou diretórios do sistema desenvolvido. Esses objetos compõem os nós da árvore de diretórios e são neles que são armazenadas todas as informações referentes aos nós que representam. São esses objetos que são serializados para serem gravados no arquivo de encapsulamento.

O TKFFS não separa os dados dos metadados, diferente da estrutura `inode` dos sistemas de arquivos comuns do Linux. A presente classe é responsável pelo armazenamento tanto dos dados de um arquivo, conteúdo de um diretório ou alvo de um *link*, quanto do metadados.

Esta classe possui os seguintes atributos:

a) `serialVersionUID`: atributo necessário para a serialização dos objetos;

b) `name`: armazena o nome do *node* representado;

c) `type`: informa se o objeto refere-se a um arquivo, diretório ou *link*;
d) `uid`: armazena o identificador do proprietário do *node* representado;
e) `gid`: armazena o identificador do grupo ao qual o *node* representado pertence;
f) `mode`: armazena as permissões do *node* representado. O seu valor inicial é **777**;
g) `size`: armazena o tamanho, caso o objeto representar um arquivo;
h) `ctime`: armazena a data e a hora da criação do *node* representado;
i) `atime`: armazena a data e a hora do último acesso ao *node* representado;
j) `mtime`: armazena a data e a hora da última modificação do *node* representado;
k) `opened`: informa se está aberto ou fechado, caso o objeto representar um arquivo;
l) `directoryContent`: armazena outros objetos desta classe, representando o conteúdo de um diretório, caso o objeto representar um diretório;
m) `fileData`: armazena matrizes de *bytes*, representando os blocos de dados, caso o objeto representar um arquivo;
n) `linkTarget`: armazena o caminho absoluto do alvo, caso o objeto representar um *link*.

Os métodos desta classe são responsáveis pela manipulação dos atributos do *node* que objeto representa. Em sua grande maioria são chamados por métodos da classe `TKFFilesystemDataSource`, para realizar as alterações necessárias. As descrições dos os métodos desta classe são apresentadas no Quadro 9.

Dentre os métodos citados, em seguida são detalhados os mais importantes, do ponto de vista da compreensão do funcionamento do sistema.

MÉTODO	DESCRIÇÃO
setName	Modifica o atributo `name`.
setType	Modifica o atributo `type`.
setUID	Modifica o atributo `uid`.
setGID	Modifica o atributo `gid`.
setMode	Modifica o atributo `mode`.
setSize	Modifica o atributo `size`.
setCTime	Modifica o atributo `ctime`.
setATime	Modifica o atributo *`atime`*.
setMTime	Modifica o atributo *`mtime`*.
setOpened	Modifica o atributo `opened`.
addDirectoryContent	Modifica o atributo `directoryContent`, adicionando um novo *node* ao seu conteúdo.
removeDirectoryContent	Modifica o atributo `directoryContent`, removendo um determinado *node* do seu conteúdo.
setLinkTarget	Modifica o atributo `linkTarget`.
getName	Retorna o conteúdo do atributo `name`.
getType	Retorna o conteúdo do atributo `type`.
getUID	Retorna o conteúdo do atributo `uid`.
getGID	Retorna o conteúdo do atributo `gid`.
getMode	Retorna o conteúdo do atributo `mode`.
getSize	Retorna o conteúdo do atributo `size`.
getNumberOfFiles	Calcula e retorna o número de arquivos que estão sob si, caso o objeto representar um diretório.
getCTime	Retorna o conteúdo do atributo `ctime`.
getATime	Retorna o conteúdo do atributo `atime`.
getMTime	Retorna o conteúdo do atributo `mtime`.
getQtdBlocks	Retorna a quantidade de matrizes de *bytes*, presentes no atributo `fileData`.
getOpened	Retorna o conteúdo do atributo `opened`.
hasDirectoryContent	Verifica se em seu conteúdo está determinado *node*, caso o objeto representar um diretório.
getDirectoryContent	Retorna um *node* presente em seu conteúdo, caso o objeto representar um diretório.
getAllDirectoryContent	Retorna todo o seu conteúdo, caso o objeto representar um diretório.
isLink	Retorna a informação que define se o objeto representa um *link*.
getLinkTarget	Retorna o alvo, caso o objeto representar um *link*.
read	Armazena no *buffer* recebido, o conteúdo das matrizes de *bytes* solicitadas, presentes no atributo `fileData`.
write	Armazena os *bytes* presentes no *buffer* recebido, em uma matriz de *bytes* e a adiciona no atributo `fileData`, na posição solicitada.
updateSize	Modifica o atributo `size`.

Quadro 9 – Métodos da classe `TKFNode`

3.3.1.3.1 Método `write`

A função deste método é modificar o conteúdo do arquivo que o objeto representa. Para isso, ele transforma o conteúdo do *buffer* recebido como parâmetro, em uma matriz de

bytes e depois a adiciona ao atributo `fileData`, com um identificador específico (`offset`).

Na linha 255, presente na Figura 31, o deslocamento, ou posição onde o conteúdo do *buffer* deve ser escrito, é transformado em um objeto da classe `Long`.

Nas linhas 257 e 258, o conteúdo do *buffer* é colocado em uma matriz de *bytes*.

Já na linha 260, o objeto da classe `Long` criado, referente ao deslocamento, serve de chave de busca para a matriz de *bytes* que é armazenada no atributo `fileData`, que é um objeto da classe `HashMap`.

Na linha 263, o método `updateSize` é chamado para atualizar o tamanho do arquivo.

```
253      public boolean write(ByteBuffer dataBuffer, long offset) {
254          try {
255              Long lg = new Long(offset);
256
257              byte[] bytes = new byte[dataBuffer.capacity()];
258              dataBuffer.get(bytes, 0, bytes.length);
259
260              fileData.put(lg, bytes);
261
262              //Atualiza o tamanho do arquivo
263              updateSize();
264
265          } catch (Exception e) {
266              e.printStackTrace();
267
268              return false;
269          }
270
271          return true;
272      }
```

Figura 31 – Método `write`

3.3.1.3.2 Método `read`

A função deste método é retornar uma parte do conteúdo do arquivo que o objeto representa, conforme solicitação.

Na linha 239, presente na Figura 32, o deslocamento, ou posição onde o conteúdo do *buffer* deve ser escrito, é transformado em um objeto da classe `Long`.

Na linha 241, uma matriz de *bytes* é obtida do atributo `fileData`, que é objeto da classe `HashMap`, tendo como chave de busca o objeto da classe `Long` criado.

Já na linha 243, a matriz *bytes* é colocada no *buffer*, para que possa ser encaminhada ao FUSE.

```
232         /*
233          * METODOS ESPECIAIS
234          */
235
236         public boolean read(ByteBuffer dataBuffer, long offset) {
237
238             try {
239                 Long lg = new Long(offset);
240
241                 byte[] bytes = (byte[])fileData.get(lg);
242
243                 dataBuffer.put(bytes, 0, bytes.length);
244             } catch (Exception e) {
245                 e.printStackTrace();
246
247                 return false;
248             }
249
250             return true;
251         }
```

Figura 32 – Método `read`

3.3.2 Operacionalidade da implementação

Para demonstrar o funcionamento do sistema de arquivos desenvolvido, são apresentados os passos necessários para instalá-lo no sistema operacional, além de casos de uso da sua utilização.

3.3.2.1 Instalação do sistema de arquivos

São pré-requisitos desta instalação, as instalações do JDK 1.5 e do FUSE. Este trabalho foi desenvolvido com base no FUSE 2.8.

Para instalar o TKFFS são necessários os passos seguintes:

a) conectar no sistema com o usuário `root`;
b) descompactar o pacote do TKFFS no diretório onde o mesmo será instalado. Para exemplificar será utilizado o diretório `/usr/src`;
c) entrar no diretório gerado pela descompactação;
d) editar o arquivo de configuração `build.conf`, colocando no conteúdo das variáveis `JDK_HOME` e `FUSE_HOME`, os diretórios referentes às instalações do JDK e do FUSE respectivamente. A Figura 33 apresenta um exemplo desta configuração;

```
#!/bin/bash
# by: Thiago Klein Flach

# JDK 1.4 or greater HOME
export JDK_HOME="/home/thiago/jdk1.5.0_20"

# FUSE library & headers base directory
export FUSE_HOME="/usr/local"
```

Figura 33 – Exemplo do conteúdo do arquivo de configuração `build.conf`

e) executar o *script* `install.sh` dentro do mesmo diretório. Este *script* irá criar o arquivo `/etc/tkffs.conf` utilizado para a configuração do TKFFS, o arquivo `/var/log/tkffs.log` para o armazenamento dos *logs*, além de compilar os códigos fonte referenciando as bibliotecas necessárias. Caso a compilação ocorra com sucesso, a mensagem "TKFFS *IS COMPILED AND BUILT!"* será exibida, como na Figura 34;

```
root@thiagotcc:/usr/src/tkffs-2009# ./install.sh
CONFIG FILE /etc/tkffs.conf CREATED
LOG FILE /var/log/tkffs.log CREATED
"/home/thiago/jdk1.5.0_20"/bin/javac -Xlint -source 1.4 -d build src/fuse/Filesy
stemToFuseFSAdapter.java src/fuse/Filesystem1ToFilesystemAdapter.java src/fuse/F
useStat.java src/fuse/FuseException.java src/fuse/FuseStatfs.java src/fuse/util/
Struct.java src/fuse/util/Log.java src/fuse/Filesystem.java src/fuse/FuseDirEnt.
java src/fuse/FuseMount.java src/fuse/FuseFS.java src/fuse/Filesystem1.java src/
fuse/FuseFSDirEnt.java src/fuse/FilesystemConstants.java src/fuse/FakeFilesystem
.java src/fuse/tkffs/util/TKFNode.java src/fuse/tkffs/TKFFilesystemDataSource.ja
va src/fuse/tkffs/TKFFilesystem.java src/fuse/FuseFtype.java src/java2c/CAPIGene
rator.java src/java2c/DumpJVMLdPath.java
make -C jni all
make[1]: Entrando no diretÃ³rio `/usr/src/tkffs-2009/jni'
make[1]: Nada a ser feito para `all'.
make[1]: Saindo do diretÃ³rio `/usr/src/tkffs-2009/jni'

TKFFS IS COMPILED AND BUILT!

root@thiagotcc:/usr/src/tkffs-2009#
```

Figura 34 – Exemplo da instalação do TKFFS no sistema operacional

f) editar o arquivo de configuração `/etc/tkffs.conf`, colocando no conteúdo das variáveis `TKFFS_HOME` e `TKFFS_LOG`, o diretório referente à instalação do TKFFS, e o caminho completo do arquivo que será utilizado para armazenar os *logs*, respectivamente, como na Figura 35.

```
#!/bin/bash
# TKFFS config file
# by: Thiago Klein Flach

# TKFFS home
export TKFFS_HOME="/usr/src/tkffs-2009"

# TKFFS log file
export TKFFS_LOG="/var/log/tkffs.log"
```

Figura 35 - Exemplo do conteúdo do arquivo de configuração `/etc/tkffs.conf`

Para facilitar o acesso aos comandos de montagem e desmontagem de um arquivo de encapsulamento do TKFFS em um diretório do sistema, pode-se criar um *link*, no diretório `/usr/bin`, para seus respectivos *scripts*, que estão no diretório de instalação do TKFFS.

A seguir são detalhados os *scrips* de controle criados para o TKFFS.

3.3.2.2 *Scripts* de controle

Neste item são detalhados os *scripts* criados para o TKFFS, referentes aos procedimentos de instalação (`install.sh`), montagem (`tkffsmount`), desmontagem (`tkffsumount`) e inicialização dos códigos Java (`tkffsengine`).

3.3.2.2.1 *Script* `install.sh`

Este *script* realiza a instalação do TKFFS no sistema operacional. As operações que ele realiza são detalhadas na Figura 36 da seguinte forma:

a) inicializa variáveis locais, referentes às localizações do arquivo de configuração (linha 4) e do arquivos de armazenamento dos *logs* (linha 5);
b) cria o arquivo de configuração (linhas 8 a 16);
c) define permissão total de acesso ao arquivo de configuração (linha 19), que também é executável;
d) informa ao usuário que o arquivo de configuração foi criado (linha 22);
e) cria o arquivo de armazenamento dos *logs* e define suas permissões (linha 25);
f) informa ao usuário que o arquivo de armazenamento dos *logs* foi criado (linha 22);
g) compila os códigos fonte do TKFFS (linha 31).

```
 1 #!/bin/bash
 2 # by: Thiago Klein Flach
 3
 4 TKFFS_CONF="/etc/tkffs.conf"
 5 TKFFS_LOG="/var/log/tkffs.log"
 6
 7 #Cria o arquivo de configuracao
 8 echo "#!/bin/bash" > $TKFFS_CONF
 9 echo "# TKFFS config file" >> $TKFFS_CONF
10 echo "# by: Thiago Klein Flach" >> $TKFFS_CONF
11 echo "" >> $TKFFS_CONF
12 echo "# TKFFS home" >> $TKFFS_CONF
13 echo "export TKFFS_HOME=\"\"" >> $TKFFS_CONF
14 echo "" >> $TKFFS_CONF
15 echo "# TKFFS log file" >> $TKFFS_CONF
16 echo "export TKFFS_LOG=\"$TKFFS_LOG\"" >> $TKFFS_CONF
17
18 #Define permissao total de acesso ao arquivo
19 chmod 777 $TKFFS_CONF
20
21 #OUTPUT
22 echo "CONFIG FILE $TKFFS_CONF CREATED"
23
24 #Tenta criar o arquivo log e definir suas permissoes
25 echo "" > $TKFFS_LOG && chmod 766 $TKFFS_LOG
26
27 #OUTPUT
28 echo "LOG FILE $TKFFS_LOG CREATED"
29
30 #Tenta compilar o codigo
31 make
```

Figura 36 – *Script* `install.sh`

3.3.2.2.2 *Script* `tkffsmount`

Este *script* realiza a montagem de um arquivo de encapsulamento do TKFFS em um diretório vazio do sistema operacional.

A sintaxe da chamada desse script é apresentada para o usuário quando ele executa `tkffsmount -h`, como na Figura 37. Neste caso `[mount_point]` é o ponto de montagem e `[base_file]` é o arquivo de encapsulamento.

```
thiago@thiagotcc:~$ tkffsmount -h
USAGE: tkffsmount [mount_point] [base_file]
thiago@thiagotcc:~$
```

Figura 37 – Sintaxe do *script* `tkffsmount`

As operações que ele realiza são detalhadas na Figura 38 da seguinte forma:

a) inicializa uma variável local, referente à localização do arquivo de configuração (linha 4) do TKFFS;
b) verifica se o arquivo de configuração existe (linha 7), caso não exista, informa o usuário (linha 57);
c) executa o arquivo de configuração do TKFFS (linha 10);
d) verifica se o arquivo de configuração foi executado com sucesso (linha 13), caso

não for, informa o usuário (linha 54);

e) verifica se o número de parâmetros recebidos é igual a 2 (linha 15), caso não for, informa o usuário sobre a sintaxe correta (linha 51);
f) verifica se o primeiro parâmetro recebido é um diretório (linha 17), caso não for, informa o usuário (linha 48);
g) se tiver / no final do caminho do diretório passado como parâmetro, remove-a e em seguida adiciona o resultado na variável `DIR` (linha 19);
h) armazena na variável `IS_MOUNTED` o número de linhas encontradas com referência ao diretório em questão, no arquivo que contém os sistemas de arquivos montados atualmente no sistema operacional (linha 22). Este número, quando positivo, indica que este diretório já está servindo como ponto de montagem para algum sistema de arquivos;
i) verifica se o diretório em questão não está sendo utilizado como ponto de montagem por algum sistema de arquivos (linha 25), caso estiver, o usuário é informado que o diretório está ocupado (linhas 44 e 45);
j) executa o *script* `tkffsengine` para inicializar a execução dos códigos Java do TKFFS em segundo plano (linha 27);
k) aguarda 4 segundos para que o arquivo de encapsulamento seja montado no diretório e disponibilizado pelo VFS (linha 30);
l) armazena na variável `IS_MOUNTED` o número de linhas encontradas com referência ao diretório em questão e ao FUSE, no arquivo que contém os sistemas de arquivos montados atualmente no sistema operacional (linha 33). Este número, quando positivo, indica que a montagem do arquivo de encapsulamento no diretório em questão ocorreu com sucesso;
m) verifica se a montagem ocorreu com sucesso (linha 36), o usuário é informado sobre isso tanto em caso negativo (linhas 39 a 41), quanto em caso positivo (linha 37).

```
#!/bin/bash
# by: Thiago Klein Flach

TKFFS_CONF="/etc/tkffs.conf"

# Se o arquivo de configuracao existe
if [ -e $TKFFS_CONF ]; then

        # Tenta executar o arquivo de configuracao
        . $TKFFS_CONF

        # Se conseguiu executar o arquivo de configuracao
        if [ 0 -eq $? ]; then
                # Se recebeu 2 parametros
                if [ 2 -eq $# ]; then
                        # Se o primeiro parametro for um diretorio
                        if [ -d $1 ]; then
                                # Remove a / do final
                                DIR=`echo $1 | sed s/"\/$"//g`

                                # Executa o comando para verificar se a montagem foi realizada
                                IS_MOUNTED=`cat /etc/mtab | grep $DIR | wc -l`

                                # Se NAO  houver algo ja mapeado no diretorio
                                if [ 0 -eq $IS_MOUNTED ]; then
                                        # Se o segundo parametro for um arquivo
                                        nohup sh $TKFFS_HOME/tkffsengine $1 $2 2>> $TKFFS_LOG &

                                        # Aguarda montar
                                        sleep 4

                                        # Executa o comando para verificar se a montagem foi realizada
                                        IS_MOUNTED=`cat /etc/mtab | grep $DIR | grep "fuse.javafs" | wc -l`

                                        # Se esta montado
                                        if [ $IS_MOUNTED -gt 0 ]; then
                                                echo "TKFFS IS MOUNTED!"
                                        else
                                                echo "TKFFS can NOT be mounted"
                                                echo "SEE: $TKFFS_LOG"
                                                tail $TKFFS_LOG
                                        fi
                                else
                                        echo "TKFFS can NOT be mounted"
                                        echo "the directory $DIR is busy"
                                fi
                        else
                                echo "$1: is NOT a directory"
                        fi
                else
                        echo "USAGE: tkffsmount [mount_point] [base_file]"
                fi
        else
                echo "the config file $TKFFS_CONF can NOT be executed"
        fi
else
        echo "the config file $TKFFS_CONF NOT exist"
fi
```

Figura 38 – *Script* `tkffsmount`

3.3.2.2.3 *Script* `tkffsumount`

Este *script* realiza a desmontagem do arquivo de encapsulamento do TKFFS, do diretório no qual ele está montado.

A sintaxe da chamada desse script é apresentada para o usuário quando ele executa `tkffsumount -h`, como na Figura 39. Neste caso `[mount_point]` é o ponto de montagem.

```
thiago@thiagotcc:~$ tkffsumount -h
usage: tkffsumount [mount_point]
thiago@thiagotcc:~$
```

Figura 39 – Sintaxe do *script* `tkffsumount`

As operações que ele realiza são detalhadas na Figura 40 da seguinte forma:

a) inicializa uma variável local, referente a localização do arquivo de configuração (linha 4) do TKFFS;
b) verifica se o arquivo de configuração existe (linha 7), caso não exista, informa o usuário (linha 55);
c) executa o arquivo de configuração do TKFFS (linha 10);
d) verifica se o arquivo de configuração foi executado com sucesso (linha 13), caso não for, informa o usuário (linha 52);
e) verifica se o número de parâmetros recebidos é igual a 1 (linha 15), caso não for, informa o usuário sobre a sintaxe correta deste comando (linha 49);
f) verifica se o parâmetro recebido é igual a `-h` (linha 17), caso for, informa o usuário sobre a sintaxe deste comando (linha 18) e finaliza;
g) verifica se o parâmetro recebido é um diretório (linha 21), caso não for, informa o usuário (linha 45);
h) se tiver `/` no final do caminho do diretório passado como parâmetro, remove e em seguida adiciona o resultado na variável `DIR` (linha 23);
i) armazena na variável `IS_MOUNTED` o número de linhas encontradas com referência ao diretório em questão e ao FUSE, no arquivo que contém os sistemas de arquivos montados atualmente no sistema operacional (linha 26). Este número, quando positivo, indica se este diretório já está servindo como ponto de montagem para o FUSE;
j) verifica se o diretório em questão não está sendo utilizado como ponto de montagem pelo FUSE (linha 29), caso não estiver, o usuário é informado (linha 42);
k) executa o comando `fusermount -u`, passando como parâmetro o diretório em questão, para que ele seja desmontado (linha 31);
l) verifica se a desmontagem ocorreu com sucesso (linha 34), o usuário é informado sobre isso tanto em caso negativo (linhas 37 a 39), quanto em caso positivo (linha 35).

```
 1 #!/bin/bash
 2 # by: Thiago Klein Flach
 3
 4 TKFFS_CONF="/etc/tkffs.conf"
 5
 6 # Se o arquivo de configuracao existe
 7 if [ -e $TKFFS_CONF ]; then
 8
 9         # Tenta executar o arquivo de configuracao
10         . $TKFFS_CONF
11
12         # Se conseguiu executar o arquivo de configuracao
13         if [ 0 -eq $? ]; then
14                 # Se recebeu um parametro
15                 if [ 1 -eq $# ]; then
16                         # Se o parametro for para mostrar o HELP
17                         if [ $1 = '-h' ]; then
18                                 echo "usage: tkffsumount [mount_point]"
19                         else
20                                 # Se o parametro for um diretorio
21                                 if [ -d $1 ]; then
22                                         # Remove a / do final
23                                         DIR=`echo $1 | sed s/"\/$"/""/g`
24
25                                         # Executa o comando para verificar se o diretorio esta montado
26                                         IS_MOUNTED=`cat /etc/mtab | grep $DIR | grep "fuse.javafs" | wc -l`
27
28                                         # Se esta montado
29                                         if [ $IS_MOUNTED -gt 0 ]; then
30                                                 # Tenta desmontar
31                                                 fusermount -u $DIR >> $TKFFS_LOG
32
33                                                 # Se foi desmontado com sucesso
34                                                 if [ 0 -eq $? ]; then
35                                                         echo "TKFFS IS UMOUNTED!"
36                                                 else
37                                                          echo "TKFFS can NOT be umounted"
38                                                          echo "SEE: $TKFFS_LOG"
39                                                          tail $TKFFS_LOG
40                                                 fi
41                                         else
42                                                 echo "TKFFS is NOT mounted in $DIR"
43                                         fi
44                                 else
45                                         echo "$1: is NOT a directory"
46                                 fi
47                         fi
48                 else
49                         echo "USAGE: tkffsumount [mount_point]"
50                 fi
51         else
52                 echo "the config file $TKFFS_CONF can NOT be executed"
53         fi
54 else
55         echo "the config file $TKFFS_CONF NOT exist"
56 fi
```

Figura 40 – *Script* `tkffsumount`

3.3.2.2.4 *Script* `tkffsengine`

Este *script* realiza a inicialização dos códigos Java do TKFFS. Ele não é chamado pelo usuário, mas sim pelo *script* `tkffsmount`. As operações que ele realiza são detalhadas na Figura 41 da seguinte forma:

a) inicializa uma variável local, referente à localização do arquivo de configuração (linha 6) do TKFFS;
b) verifica se o arquivo de configuração existe (linha 9), caso não exista, informa o usuário (linha 31);
c) executa o arquivo de configuração do TKFFS (linha 12);

d) verifica se o arquivo de configuração foi executado com sucesso (linha 15), caso não for, informa o usuário (linha 28);

e) executa o arquivo de configuração `build.conf` (linha 49);

f) verifica se o arquivo de configuração foi executado com sucesso (linha 21), caso não for, informa o usuário (linha 25);

g) inicializa o TKFFS pela classe `TKFFilesystem`, passando como parâmetros o ponto de montagem e o a localização do arquivo de encapsulamento (linha 23).

```
 1 #!/bin/sh
 2 # by: Thiago Klein Flach
 3
 4 # Usage: ./tkffs_mount.sh /mount/point file
 5
 6 TKFFS_CONF="/etc/tkffs.conf"
 7
 8 # Se o arquivo de configuracao existe
 9 if [ -e $TKFFS_CONF ]; then
10
11         # Tenta executar o arquivo de configuracao
12         . $TKFFS_CONF
13
14         # Se conseguiu executar o arquivo de configuracao
15         if [ 0 -eq $? ]; then
16
17                 # Tenta executar o arquivo build.conf
18                 . $TKFFS_HOME/build.conf
19
20                 # Se conseguiu executar
21                 if [ 0 -eq $? ]; then
22
23                         LD_LIBRARY_PATH=$TKFFS_HOME/jni:$FUSE_HOME/lib $JDK_HOME/bin/java -classpath
   $TKFFS_HOME/build fuse.tkffs.TKFFilesystem -f -s $1 $2
24                 else
25                         echo "the file $TKFFS_HOME/build.conf can NOT be executed"
26                 fi
27         else
28                 echo "the config file $TKFFS_CONF can NOT be executed"
29         fi
30 else
31         echo " the config file $TKFFS_CONF NOT exist"
32 fi
```

Figura 41 – *Script* `tkffsengine`

3.3.2.3 Utilização do sistema de arquivos

A utilização do sistema de arquivos se dá por meio da montagem e desmontagem de um arquivo de encapsulamento em um diretório vazio do sistema operacional, utilizando os *scripts* já citados. É importante observar que o TKFFS não aceita ser montado em diretórios que já possuem algum conteúdo.

Neste item são apresentados dois casos de uso da utilização do TKFFS.

No primeiro caso, o usuário monta um novo arquivo de encapsulamento em um diretório vazio do sistema operacional e realiza as seguintes operações (Figura 42):

a) acessa o interior do arquivo de encapsulamento, através do ponto de montagem;

b) lista o conteúdo;
c) cria dois diretórios;
d) lista o conteúdo;
e) acessa o primeiro diretório criado;
f) cria um arquivo dentro do diretório;
g) lista o conteúdo do diretório;
h) muda as permissões do arquivo criado;
i) lista o conteúdo do diretório;
j) insere algumas linhas de texto no arquivo;
k) mostra o conteúdo do arquivo;
l) acessa o segundo diretório criado;
m) cria um *link* dentro do diretório, apontando para o arquivo do primeiro diretório;
n) lista o conteúdo do diretório;
o) mostra o conteúdo do arquivo através do *link*;
p) executa um comando para obter apenas a última linha do arquivo, através do *link*;
q) sai do ponto de montagem;
r) desmonta o arquivo de encapsulamento.

No segundo caso, o usuário monta o arquivo de encapsulamento criado no primeiro, no mesmo ponto de montagem e realiza as seguintes operações (Figura 43):

a) executa a descompactação de um arquivo TAR (*Tape Archive*), de fora do ponto de montagem, redirecionando a saída para o seu interior;
b) acessa o interior do arquivo de encapsulamento, através do ponto de montagem;
c) lista o conteúdo;
d) obtém o tamanho de todos os arquivos e diretórios presentes na raiz do sistema de arquivos montado;
e) remove um diretório com todo seu conteúdo;
f) lista o conteúdo da raiz do sistema de arquivos;
g) sai do ponto de montagem;
h) tenta remontar o arquivo de encapsulamento no mesmo ponto de montagem;
i) desmonta o sistema de arquivos;
j) verifica o tamanho com que ficou o arquivo de encapsulamento.

```
thiago@thiagotcc:~$ tkffsmount /home/thiago/mnt tkffsfile
TKFFS IS MOUNTED!
thiago@thiagotcc:~$ cd /home/thiago/mnt/
thiago@thiagotcc:~/mnt$ ls -lah
total 0
thiago@thiagotcc:~/mnt$ mkdir diret1
thiago@thiagotcc:~/mnt$ mkdir diret2
thiago@thiagotcc:~/mnt$ ls -lah
total 0
drwxr-xr-x 0 root root 0 2009-11-20 15:31 diret1
drwxr-xr-x 0 root root 0 2009-11-20 15:31 diret2
thiago@thiagotcc:~/mnt$ cd diret1
thiago@thiagotcc:~/mnt/diret1$ touch arq
thiago@thiagotcc:~/mnt/diret1$ ls -lah
total 0
-rw-r--r-- 0 root root 0 2009-11-20 15:32 arq
thiago@thiagotcc:~/mnt/diret1$ chmod 664 arq
thiago@thiagotcc:~/mnt/diret1$ ls -lah
total 0
-rw-rw-r-- 0 root root 0 2009-11-20 15:32 arq
thiago@thiagotcc:~/mnt/diret1$ vi arq
thiago@thiagotcc:~/mnt/diret1$ cat arq
Linha 1
Linha 2
Linha 3
thiago@thiagotcc:~/mnt/diret1$ cd ../diret2
thiago@thiagotcc:~/mnt/diret2$ ln ../diret1/arq lin
thiago@thiagotcc:~/mnt/diret2$ ls -lah
total 512
-rw-rw-r-- 0 root root 24 2009-11-20 15:32 lin
thiago@thiagotcc:~/mnt/diret2$ cat lin
Linha 1
Linha 2
Linha 3
thiago@thiagotcc:~/mnt/diret2$ tail -1 lin
Linha 3
thiago@thiagotcc:~/mnt/diret2$ cd ../..
thiago@thiagotcc:~$ tkffsumount /home/thiago/mnt
TKFFS IS UMOUNTED!
thiago@thiagotcc:~$
```

Figura 42 – Exemplo de utilização do TKFFS

```
thiago@thiagotcc:~$ tkffsmount /home/thiago/mnt tkffsfile
TKFFS IS MOUNTED!
thiago@thiagotcc:~$ tar -xzvf tkffs-2009.tgz -C /home/thiago/mnt
tkffs-2009/
tkffs-2009/src/
tkffs-2009/src/fuse/
tkffs-2009/src/fuse/FilesystemToFuseFSAdapter.java
tkffs-2009/src/fuse/Filesystem1ToFilesystemAdapter.java
tkffs-2009/jni/javafs_bindings.o
(...)
tkffs-2009/build.conf
tkffs-2009/tkffsengine
tkffs-2009/tkffsumount
tkffs-2009/tkffsmount
tkffs-2009/nohup.out
tkffs-2009/fuse.iws
thiago@thiagotcc:~$ cd /home/thiago/mnt
thiago@thiagotcc:~/mnt$ ls -lah
total 0
drwxr-xr-x 0 root root   24 2009-11-20 15:31 diret1
drwxr-xr-x 0 root root    0 2009-11-20 15:31 diret2
drwxr-xr-x 0 root root 446K 2009-11-20 08:00 tkffs-2009
thiago@thiagotcc:~/mnt$ du -sh *
512     diret1
512     diret2
97K     tkffs-2009
thiago@thiagotcc:~/mnt$ rm -rf diret2
thiago@thiagotcc:~/mnt$ ls -lah
total 0
drwxr-xr-x 0 root root   24 2009-11-20 15:31 diret1
drwxr-xr-x 0 root root 446K 2009-11-20 08:00 tkffs-2009
thiago@thiagotcc:~/mnt$ cd ..
thiago@thiagotcc:~$ tkffsmount /home/thiago/mnt tkffsfile
TKFFS can NOT be mounted
the directory /home/thiago/mnt is busy
thiago@thiagotcc:~$ tkffsumount /home/thiago/mnt
TKFFS IS UMOUNTED!
thiago@thiagotcc:~$ ls -lah tkffsfile
-rw-r--r-- 1 thiago thiago 459K 2009-11-20 15:52 tkffsfile
thiago@thiagotcc:~$
```

Figura 43 – Exemplo de utilização do TKFFS

3.4 RESULTADOS E DISCUSSÃO

Após a implementação do TKFFS ter sido concluída, foram realizados alguns testes focando principalmente nas questões de desempenho, confiabilidade e independência do dispositivo utilizado como ponto de montagem.

Para testar a questão do desempenho, foram comparados os tempos necessários para o processo de descompactação de um arquivo de 22 MB em um disco rígido, através de um diretório comum do sistema operacional (Figura 44) e em um arquivo de encapsulamento do TKFFS, através do ponto de montagem (Figura 45). No final das execuções apresentadas nas figuras 44 e 45, está o tempo total do consumo de CPU pelo processo de descompactação (**real**), o tempo necessário para o processamento das chamadas realizadas pelo usuário (**user**) e o tempo necessário para o processamento das chamadas de sistema (**sys**).

```
thiago@thiagotcc:~$ time unzip android-sdk_r04-windows.zip -d unpackdir
Archive:  android-sdk_r04-windows.zip
   creating: unpackdir/android-sdk-windows/
   creating: unpackdir/android-sdk-windows/platforms/
  inflating: unpackdir/android-sdk-windows/SDK Readme.txt
  inflating: unpackdir/android-sdk-windows/SDK Setup.exe
   creating: unpackdir/android-sdk-windows/tools/
  inflating: unpackdir/android-sdk-windows/tools/draw9patch.bat
  inflating: unpackdir/android-sdk-windows/tools/Jet/JetCreator/img_Open.py
  (...)
   creating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/
  inflating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/README.txt
   creating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/JET_Logic8.logic/
   creating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/JET_Logic8.logic/LgDoc/
  inflating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/JET_Logic8.logic/LgDoc/displayState
  inflating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/JET_Logic8.logic/LgDoc/documentData
  inflating: unpackdir/android-sdk-windows/tools/sqlite3.exe
   creating: unpackdir/android-sdk-windows/add-ons/

real    0m2.898s
user    0m0.704s
sys     0m1.064s
thiago@thiagotcc:~$
```

Figura 44 – Tempo para a descompactação em um dispositivo de armazenamento comum

```
thiago@thiagotcc:~$ tkffsmount mnt tkffsfile
TKFFS IS MOUNTED!
thiago@thiagotcc:~$ time unzip android-sdk_r04-windows.zip -d mnt
Archive:  android-sdk_r04-windows.zip
   creating: mnt/android-sdk-windows/
   creating: mnt/android-sdk-windows/platforms/
  inflating: mnt/android-sdk-windows/SDK Readme.txt
  inflating: mnt/android-sdk-windows/SDK Setup.exe
   creating: mnt/android-sdk-windows/tools/
  inflating: mnt/android-sdk-windows/tools/draw9patch.bat
  (...)
   creating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/
  inflating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/README.txt
   creating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/JET_Logic8.logic/
   creating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/JET_Logic8.logic/LgDoc/
  inflating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/JET_Logic8.logic/LgDoc/displayState
  inflating: unpackdir/android-sdk-windows/tools/Jet/logic_templates/JET_Logic8.logic/LgDoc/documentData
  inflating: unpackdir/android-sdk-windows/tools/sqlite3.exe
   creating: unpackdir/android-sdk-windows/add-ons/

real    0m5.698s
user    0m0.544s
sys     0m1.144s
thiago@thiagotcc:~$ tkffsumount mnt
TKFFS IS UMOUNTED!
thiago@thiagotcc:~$
```

Figura 45 – Tempo para a descompactação em um arquivo de encapsulamento do TKFFS

Como visto nas figuras acima, há um significativo aumento no tempo total necessário para o processamento da descompactação em um arquivo de encapsulamento do TKFFS, em relação a um dispositivo de armazenamento comum. Isso se deve principalmente por causa da criação de um novo objeto da classe `TKFNode` para cada arquivo e diretório criado, com a consequente manipulação de seu conteúdo, antes de sua inserção na árvore de diretórios.

O TKFFS consome a quantidade de recursos comuns a um processo do Java. Durante as descompactações apresentadas, foi notado um consumo máximo de 45% da CPU e 7.5% da memória disponível. Há de se considerar que os testes foram feitos em uma máquina virtual, com recursos limitados e que o consumo de memória é diretamente proporcional ao tamanho do conteúdo do sistema de arquivos.

Para testar a questão da confiabilidade, foi realizada a exclusão do arquivo de encapsulamento no momento em que ele está montado no sistema operacional e que o usuário está manipulando o seu conteúdo. Em casos como este não há perda de dados, pois o procedimento de montagem do TKFFS carrega o nome do arquivo e todo o seu conteúdo para a memória e então o libera, não o mantendo aberto, assim sendo, mesmo que o arquivo for removido o seu conteúdo pode ser manipulado normalmente. No momento da desmontagem, o TKFFS serializa o objeto que representa todo o conteúdo do sistema de arquivos e grava o resultado no arquivo com o nome utilizado na montagem, independente do fato de ele existir.

A questão da independência do dispositivo utilizado como ponto de montagem, foi testada com a montagem do TKFFS em um diretório presente no interior de um *pen drive* e em outro diretório presente no interior de um compartilhamento de uma outra máquina da rede. O FUSE permite que isso seja feito, pois ao receber a requisição do VFS e verificar que o caminho do arquivo, diretório ou *link* manipulado está sob o ponto de montagem, realiza os procedimentos de manipulação do arquivo de encapsulamento, não efetuando alteração alguma no sistema de arquivos original do dispositivo.

4 CONCLUSÕES

O trabalho de Mizaki, Diez e Lundberg (2007) é de grande valia em redes onde se deseja que o armazenamento seja feito de forma redundante e oportunista, pois os dados do sistema de arquivos em questão são espelhados em várias máquinas e os blocos são armazenados nas máquinas conforme suas disponibilidades de recursos.

Já o trabalho de Neto (2006) possibilita a atualização de *blogs* através de um sistema de arquivos instalável, onde os artigos são apresentados na forma que arquivos que podem ser normalmente editados. Este trabalho permitiu a compreensão da funcionalidade do FUSE-J.

O ClamFS desenvolvido por Burghardt (2009), integra a interface de um sistema de arquivos comum com o antivírus ClamAV, que realiza a varredura por vírus nos arquivos em tempo de execução.

Entretanto, nenhum dos sistemas de arquivos acima realiza um encapsulamento de seus arquivos possibilitando facilitar o transporte dos mesmos para outra máquina, ou para a realização de cópias de segurança, como neste trabalho.

Como demonstrado, a criação do TKFFS foi realizada através da adoção do FUSE-J como substrato.

Superadas as dificuldades iniciais de compreensão do funcionamento da biblioteca, o desenvolvimento transcorreu normalmente.

Até onde foi possível pesquisar, o presente trabalho contempla um sistema de arquivos completo que pode ser desmontado em uma máquina e montado em outra que tenha suporte ao FUSE, ao Java e que possua o TKFFS devidamente instalado.

Este trabalho apresenta um novo sistema de arquivos que tem como principal objetivo o encapsulamento de arquivos. O encapsulamento gerado pelo sistema de arquivos que este trabalho propõe, garante facilidade na manipulação de seu conteúdo, sendo necessário apenas montar o arquivo em questão em um diretório do sistema, para que então seja possível navegar e manipular seus arquivos e diretórios, como se ele fosse um diretório comum do sistema operacional.

Uma das limitações do sistema de arquivos desenvolvido neste trabalho é que ele armazena tudo em memória, o que limita o seu tamanho ao número de memória disponível no sistema operacional. Por outro lado, o fato do carregamento de todo o conteúdo para a memória torna-se uma virtude quando analisado do ponto de vista da segurança dos dados.

O TKFFS é um sistema de arquivos seguro, pois os procedimentos de manipulação do

arquivo de encapsulamento evitam a corrupção do mesmo. Como todo o conteúdo é manipulado em memória e o arquivo de encapsulamento não permanece aberto durante a manipulação, a ocorrência de um desligamento não programado da máquina durante a manipulação do conteúdo causa a perda apenas do que foi manipulado após a montagem, o que havia no arquivo de encapsulamento antes do carregamento do seu conteúdo para a memória permanece intacto. Além disso, a remoção do arquivo de encapsulamento após a sua montagem em um diretório do sistema operacional, não caracteriza a perda de todo o conteúdo, pois no momento da montagem todo o seu conteúdo é carregado para a memória e no momento da desmontagem o conteúdo é gravado em um arquivo com o mesmo nome.

4.1 EXTENSÕES

Como foi visto, para poder manipular o arquivo de encapsulamento, o TKFFS armazena todo o seu conteúdo na memória, após carregá-lo. Isso limita o tamanho total do sistema de arquivos ao número de memória disponível no sistema operacional.

Sugere-se implementar um método de armazenamento em blocos, dentro do arquivo, o que possibilitaria que a leitura e a escrita fossem realizadas carregando para a memória apenas o que fosse necessário.

Também é sugerido o desenvolvimento de um método para a compactação dos arquivos inseridos no TKFFS, em tempo de execução e de outro método para a descompactação dos mesmos ao serem extraídos.

Outra sugestão é a implementação de um método que criptografe o conteúdo do arquivo de encapsulamento.

Sugere-se também a implementação de um método para possibilitar que o conteúdo de um arquivo de encapsulamento possa ser migrado entre duas máquinas ligadas em rede e que possuam o TKFFS devidamente instalado. Tal migração pode ser realizada com a transmissão dos dados de forma serializada, sobre o protocolo TCP/IP.

REFERÊNCIAS BIBLIOGRÁFICAS

BURGHARDT, K. **ClamFS**. [S.l.], 2009. Disponível em: <http://clamfs.sourceforge.net/>. Acesso em: 31 out. 2009.

CARVALHO, Roberto P. **Sistema de arquivos paralelos**: alternativas para a redução do gargalo no acesso ao sistema de arquivos. 2005. 131 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.

DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R. **Sistemas operacionais**. 3. ed. Tradução Arlete Simille Marques. São Paulo: Pearson Education do Brasil, 2005. 760 p.

FUSE. **Filesystem in userspace**. [S.l.], [2009]. Disponível em: <http://fuse.sourceforge.net/>. Acesso em: 31 out. 2009.

FUSE-J. **Project web hosting**: open source software. [S.l.], [2009]. Disponível em: <http://fuse-j.sourceforge.net/>. Acesso em: 31 out. 2009.

JONES. T. **Anatomia do sistema de arquivos do Linux**. Longmont, 2007. Disponível em: <http://www.ibm.com/developerworks/br/library/l-linux-filesystem/index.html>. Acesso em: 12 nov. 2009.

LEITE, T. **Tipos de arquivos em sistemas Unix-like**. [S.l.], 2008. Disponível em: <http://localdomain.wordpress.com/2008/02/05/tipos-de-arquivos-em-sistemas-unix-like>. Acesso em: 31 out. 2009.

LOVE, R. **Desenvolvimento do kernel do Linux**. Tradução Eveline Vieira Machado. São Paulo: Ciência Moderna, 2004. 355 p.

MACHADO, F. B.; MAIA, L. P. **Arquitetura de sistemas operacionais**. 4. ed. Rio de Janeiro: LCT, 2007. 324 p.

MIAZAKI, D. T. P.; DIEZ, M.; LUNDBERG, R. U. **MOOFS mobile objects oriented file system**. 2007. 47 f. Trabalho de Formatura Supervisionado (Bacharelado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.

NETO, E. S. **Mundaú – distributed computing (or not)**: bloggerFS. [S.l.], 2006. Disponível em: < http://mundau.blogspot.com/2006/12/bloggerfs-last-night-i-was-playing.html>. Acesso em: 31 out. 2009.

OMAKE. **The Fuse binding**. [S.l], [2009]. Disponível em: <http://omake.metaprl.org/prerelease/omake-dll-fuse.html>. Acesso em: 31 out. 2009.

PFENNING, F. **File systems and FUSE**. Pittsburgh, [2009?]. Disponível em: <http://www.cs.cmu.edu/~fp/courses/15213-s07/lectures/15-filesys/index.html>. Acesso em: 31 out. 2009.

POSSAMAI, C. S. **Protótipo de gerenciador de arquivos para ambiente distribuído**. 2000. 103 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

TANENBAUM, Andrew S.; WOODHULL, Albert S. **Sistemas operacionais**: projeto e implementação. 2. ed. Tradução Edson Furmankiewicz. Porto Alegre: Bookman, 2000.

WATSON, D. **Linux daemon writing howto**. [S.l.], 2004. Disponível em: <http://www.netzmafia.de/skripten/unix/linux-daemon-howto.html>. Acesso em: 14 nov. 2009.