# BrisaFS - Um sistema de arquivos baseado em FUSE

**Antonio Carlos Marchandt Zidoi**[1], **Guilherme Henrique Vermiglio**[1], **Matheus da Silva Ferreira**[1], **Matheus dos Santos Pereira**[1]

[1]Centro de Matemática, Cognição e Computação
Universidade Federal do ABC (UFABC)
Av. dos Estados, 5001 - Bangú – Santo André – SP – Brazil

{antonio.zidoi, guilherme.vermiglio,

{matheus.ferreira, matheus.s}@aluno.ufabc.edu.br

***Abstract.*** *This project was built in conjunction with the C programming language, where a set of FUSE (Filesystem in Userspace) calling tools was implemented. We have been offered a type of premature operating system, with the purpose of supporting software and hardware to meet the minimum requirements for system operation.*

***Resumo.*** *Este projeto apresenta uma implementação realizada através da linguagem de programação C, onde foi utilizado umas de suas famosas bibliotecas chamada de FUSE (Filesystem in Userspace). Nos foi disponibilizado um tipo de sistema operacional prematuro, logo foi realizado diversos ajustes técnicos a nível de software e hardware para atender a requisitos mínimos para funcionamento desse sistema.*

## 1. Introdução

Filesystem in Userspace (FUSE) é uma interface em software para sistemas operacionais estilo UNIX que permite que usuários sem direitos de administração criem seus próprios sistemas de arquivos sem a necessidade de editar código do kernel do sistema operacional. Isso é feito através da execução do código do sistema de arquivos em modo do usuário conversando com o módulo FUSE, este sim executando em modo privilegiado, fazendo a ponte com as interfaces do sistema operacional propriamente dito.

## 2. Instruções de Compilação e Execução

Para criar o seu executável do seu código digite no terminal:

```
gcc -D_FILE_OFFSET_BITS=64 -Wall -Werror -O3 -g brisafs_v2.c
-o brisafs -lfuse
```

Crie uma pasta para simular o seu sistema de arquivos

```
mkdir /tmp/brisa_mnt_point
```

Depois iniciar o sistema de arquivos:

```
./brisafs -f -d -s /tmp/brisa_mnt_point
```

Obs.: Ao compilar o código, utilizamos 10% da memoria do computador, ou aproximadamente 1,4Giga (teto definido pelo grupo).

Obs.: Tivemos problemas referentes à falha de segmentação de memória quando tentávamos usar muita memoria várias vezes seguidas (derrubando e subindo o sistema de arquivos). Por conta disso, foi definido um valor mais baixo de memória que cumpre todos os requisitos dos *milestones*, em caso de falha de segmentação de memoria, limpe a memoria RAM, ou reinicie o computador.

## 3. Descrição do Projeto

A seguir, será descrito o que foi implementado no Projeto, usando como guia os tópicos do *milestones*. O código apresentado no github está comentado com observações referentes à implementação. Serão descritos aqui aspectos mais gerais e explicações da logica do Sistema de Arquivos projetado pelo grupo.

### 3.1. Funcionamento do sistema de arquivos

O sistema de arquivos é dinâmico e depende da memória RAM total do computador. Esta informação foi obtida por meio da chamada de sistema ”cat /proc/meminfo”. Após a tentativa de usar 10%, foi definido um teto de inferior a 1,4GB para evitar problemas durante a compilação, este teto foi escolhido de modo a cumprir todos os *milestones*.

O programa é iniciado reservando a memória descrita no paragrafo acima, atribuindo a todos os valores dela para 0, evitando assim lixo. Conforme o uso, são preenchidos os inodes. Assim, sabe-se que todos os inodes com o bloco = 0 estão livres.

Para poder armazenar qualquer tamanho de arquivo é usadas a informação bloco, que fala em qual posição do nosso bloco começa o arquivo, e a quant_blocos, que fala quantos blocos aquele arquivo ocupa. Assim, é possível guardar arquivos de qualquer tamanho, desde que não ultrapassem a memória disponível, caso isso ocorra haverá um erro na gravação.

Foi usada uma lógica de escrita linear. Como não há paralelismo neste projeto, foi utilizado um contador ”gravacao_bloco_conteudo” para informar onde a escrita está sendo feita, e quando dele um arquivo reorganizo os dados(superbloco) e conteúdos (blocos de conteúdo) para preservar esta minha ordem de escrita.

Quando vou fazer a persistência, apenas gravo os blocos ocupados de conteúdo e todo o superbloco, assim garanto um nível de eficiência.

Sempre que inicio o sistema de arquivos, ele procura se existe o arquivo de persistência, caso ele exista, ele carrega o arquivo na memoria RAM (que já zerada para evitar lixo), e assim continua normal, ou criar ou excluir o arquivo, reescreve a persistencia. É criado pelo nosso sistema um arquivo binário persistencia.

Colocamos uns debug no código para que ele escreve no terminal onde ele esta escrevendo exatamente nos blocos, e quantos bloco o arquivo gravado está usando.

### 3.2. Estrutura de dados utilizadas

Como principio básico da manipulação de arquivos, foi utilizado uma estrutura (struct) chamada de Inode, que em suma é responsável por guardar as informações de diretórios e arquivos do sistema operacional, que é sempre atualizado quando os dados forem alterados pelo usuário no computador.

Informações do Inode são por exemplo, a respeito de data e hora de modificação, tamanho, direitos de acesso, dentre outros. Segue abaixo a ”struct” responsável por gravar essas informações no sistema, após ajuste a adição de novas funcionalidades.

```
typedef struct {
    char nome[250];
    uint16_t direitos;
    long int tamanho;
    uid_t usuario;
    gid_t grupo;
    time_t last_access;
    time_t last_mod;
    uint16_t bloco;
    int quant_blocos;
} inode;
```

Todo o tipo de chamada que for a respeito de operações em sistema de arquivos irá retornar uma chamada de sistema Unix tecnicamente chamada de ”stat()” retornando assim atributos do arquivo. O Sistema Operacional através dos Inodes, permitir localizar em quais blocos estão os dados de cada arquivo e também ter um controle de quais blocos estão livres.

### 3.3. Modificações realizadas

Para atender aos objetivos do projeto, foram feitas alterações sobre o código original apresentado, a fim de obter um sistema operacional que efetue as principais operações básicas necessárias para sua utilização.

#### 3.3.1. Persistência

O nosso sistema de arquivos foi construído com base no do professor Emílio, que utiliza a memoria RAM para simular o sistema. Assim, todo o processo ocorre na memoria RAM.

Como o sistema de arquivos é simulado na memoria RAM, ele não possui persistência, ou seja, quando desligamos o sistema tudo que foi feito é perdido.

Para sanar este problemas escrevemos dois métodos que fazem read e write em um arquivo binário, estes métodos armazenam e recuperam todas as informações do nosso sistema de arquivos.

#### 3.3.2. Armazenamento e recuperação de datas

Para parametrizar as informações de último acesso e última modificação foram utilizadas duas variáveis para o processo, onde as informações coletadas foram recebidas através do *array* ”timespec ts[]”.

Após receber os valores do *array*, os dados foram disponibilizados na função *getattr*.

#### 3.3.3. Armazenamento e alteração de direitos dos usuários

Os direitos de acesso e alteração de arquivos são armazenados pelo campo ”direitos”, inserido no inode e no arquivo dentro do superbloco do brisafs. Todos os arquivos são criados com o direito padrão (ou modo) 644, que habilita os direitos de leitura e escrita

ao usuário proprietário do arquivo e direito de leitura aos outros usuários.

Também foram adicionados os campos ”usuario” e ”grupo” na estrutura do inode e no superbloco, para armazenar usuário e grupo de usuários proprietário dos arquivos. Por padrão, a qualquer arquivo criado são atribuídos usuário e grupo *root*.

Por fim, foram implementadas as funções *chown_brisafs*, *chgrp_brisafs* e *chmod_brisafs*, que permitem alterar usuario e grupo proprietário de um arquivo, além das permissões (modo) no superbloco.

### 3.3.4. Aumento do número máximo de arquivos

Na estrutura inicial do sistema de arquivos o superbloco ocupava o primeiro bloco, criando um limite para a quantidade de arquivos, dados que todos os inode estão no superbloco (1 inode = 1 arquivo), para podemos aumentar a quantidade de arquivos tivemos duas escolhas: primeira aumentar o tamanho do bloco e a segunda é fazer com que superbloco ocupe vários blocos.

Optamos pela segunda opção, para isso fazemos com que o sistema separa 1% da memória disponível para o programa para conter o superbloco. Deste modo, temos um valor dinâmico referente a quantidade de arquivos máximos, que depende da quantidade de memoria alocada para o programa.

No caso o projeto exige pelo menos 1024 arquivos, o computador que roda no teto definido para o projeto, 1,4G de espaço reservado, suporta 3500 arquivos. Porém vale lembrar que o numero é dinâmico, se retirar a restrição de memoria irá aguentar qualquer quantidade de arquivos desde que haja memória para suportá-lo.

### 3.3.5. Suporte à criação de diretórios

Não foi implementado.

### 3.3.6. Exclusão de arquivos

Para excluir os arquivos, primeiro guardamos a quantidade de blocos que o arquivo deletado está ocupando, depois zeramos todos os dado do inode do arquivo que foi apagado, depois disso movemos todos os ponteiro dos inode a direita do inode apagado para que apontem para o novo local do conteúdo, e depois movemos o conteúdo.

Assim tanto os nossos inodes no superbloco, quanto os blocos de conteúdo, sempre serão linear e terão a relação de ordem igual, garantindo assim que não precisamos nos preocupar com os dados anteriores do inode apagado, nem no superbloco nem nos blocos de conteúdo.

### 3.3.7. Suporte a discos de tamanhos arbitrários

Usamos uma chamada de sistema para montar isso, está comentado no código e na referencia a fonte, modificamos ela para pegar a memoria total de RAM do computador e usar 10% dela como memoria do sistema de arquivos, porem por motivos de

implementação, decidimos colocar um teto na quantidade de memoria, sendo que este teto cumpre todos os requisitos pedido nos *milestones*, que é poder armazenar mais que 1G e 1024 arquivos, sendo que o teto compreende 1,335GB de tamanho para os arquivos e 3500 arquivos no máximo.

#### 3.3.8. Controle de arquivos abertos/fechados

Não foi implementado

### 3.4. Funcionalidades excepcionais

Não foram incluídas funcionalidades adicionais

## 4. Conclusão

Após concluído o projeto, chegamos a um consenso de que o entendimento e implementação foi bastante complexo e difícil. A implementação se tornou demasiadamente grande e difícil para debugar por conta do fuse. Além disso, foi extremamente escasso a quantidade de informação a respeito compartilhado na internet, tornando difícil encontrar informações úteis. Por dependermos do Hardware, encontramos diversos problemas de alocação de memória que por vezes travam o computador, impedindo o avanço no projeto. No geral, obtivemos um grande aprendizado na utilização e manipulação de ponteiros e aceso de memória, observando sua importância para um sistema. Ainda foi colocado em pratica alguns tópicos da matéria sobre Sistemas Operacionais, principalmente sobre estrutura de dados e gerenciamento de arquivos.

## References


Ibm funções. `https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/rtref/fopen.htm`. Acessado: 2019-05-05.

libfuse api documentation. `https://libfuse.github.io/doxygen/index.html`. Acessado: 2019-05-05.

Oveleaf latex. `https://www.overleaf.com/`. Acessado: 2019-05-05.

Repositorio github. `https:https://github.com/`. Acessado: 2019-05-05.

Stackoverflow. `https://stackoverflow.com/questions/349889/how-do-you-determine-the-amount-of-linux-system-ram-in-c`. Acessado: 2019-05-05.

[lib ] [sta ] [git ] [ove ] [ibm ]