Bruno Aristimunha, Eric Shimizu Karbstein, Jair Edipo, Gabriel Martins Trettel

1 Descrição do projeto

Esse trabalho apresenta um sistema de arquivo para espaço de usuário, denominado *Chaos File System* - ChaosFS. Em suma, sistemas de arquivos são gerenciados pelo sistema operacional e permitem o armazenamento, recuperação e manipulação de dados. Esse sistema permite a organização de dados e metadados em um escopo limitado, detalhado na Seção 3.

Para tanto empregamos a interface em software *Filesystem in Userspace*, também conhecida como FUSE. Essa biblioteca nos possibilita realizar a comunicação com núcleo do Linux, sem a necessidade de permissões adicionais, re-compilação ou edição do *Kernel*.

Através desse intermediador implementamos funções relacionadas ao sistema de arquivos. Essa implementação é mapeada para funções já definidas da interface e possibilita a comunicação com o Sistema Operacional.

2 Modificações feitas

As modificações são descritas em quatro sub-seções e um quadro, sendo elas: re-estruturação do código inicial, suporte a armazenamento do tempo de acesso/modificação, suporte aos proprietários dos arquivos, aumento do limite de arquivos criados e o quadro com a descrição de cada método mapeado para o FUSE, descrito na Tabela 1.

2.1 Reestruturação

Modificamos toda ocorrência de "brisafs"para o nome do *filesystem* deste projeto, ou seja, "chaosfs". Também houve a modularização do código, separando-o em vários arquivos:

- chaosfs.c: Onde fica a chamada da função *main* e a lógica principal das funções necessárias passadas pro fuse.
- chaosfs.h: Armazena as definições de funções do **chaosfs.c** e inclusão de bibliotecas usada por ele.
- commom.h: O local dos *defines* e da definição do *struct* do *inode* e da inclusão de todas as bibliotecas compartilhadas entre os outros arquivos.
- utils.c: Implementação de funções auxiliares que não possuem dependência direta com estruturas do filesystem.
- utils.h: Definição das funções de **utils.h** e inclusão de bibliotecas usadas somente nele.
- Makefile: Um *makefile* para facilitar o processo de compilação e limpeza dos arquivos gerados pelo código fonte.

2.2 Suporte a armazenamento do tempo de acesso e modificação

Para conseguir armazenar estes parâmetros, tivemos que modificar a estrutura do *inode*, adicionando os campos **data_acesso** e **data_modific** do tipo *time_t*, representando a última

Tabela 1: Quadro de funções desenvolvidas com descrição sobre as modificações realizadas para funcionado e especificação no ChaosFS

Métodos	**Descrição**
getattr	Esta função é chamada por programas como ls, para recuperar meta-informações sobre os arquivos dentro do FS para o usuário. Nossas modificações neste trecho de código se resumiram a adicionar ao *struct* retornado ao FUSE as outras informações que estamos armazenando, como a data.
fsync	A função é executada quando um programa tenta sincronizar seu *buffer* de escrita, em memória principal, no arquivo armazenado no sistema de arquivos que o contém. A implementação desta funcionalidade se resume numa sub-rotina que abre o arquivo em disco e cópia nele todo o *file system (FS)* em memória.
utimens	É a função que atualiza a data de um arquivo, com precisão de nano-segundos. Sua implementação é simples e se resume à encontrar o bloco contendo o inode do arquivo que desejamos modificar suas meta-informações e, quando encontrado, alteramos o conteúdo do campo que armazena a data para o valor atual.
destroy	A função destroy é chamada quando o processo do sistema de arquivos é cancelada pelo usuário, e nela espera-se executar algum tipo de tarefa para manter a consistência dos dados e nenhuma informação se perder no caminho. Nossa implementação apenas verifica se existem dados ainda não salvos, e caso sim, os salva.
write	Esta função é chamada pelo FUSE quando um arquivo precisa ser escrito no FS. As modificações neste método se resumem a escrever no campo data_modific do inode a data de modificação do arquivo que está sedo escrita. Outra modificação neste método é na implementação da persistência dos dados; a cada 20 chamadas dessa função, o próprio FS executa o `fsync` para persistir os dados em disco. A estratégia de não escrever o disco a cada nova escrita é por motivos de performance, já que no ChaosFS, todo o FS é re-escrito a cada chamada do *fsync*.
rmdir	É a função chamada para deletar um diretório existente
unlink	É a função chamada para remover qualquer arquivo que não seja do tipo diretório
chmod	A função é chamada quando o comando de terminal *chmod* é chamado para alterar as permissões de leitura e escrita do arquivo
chwon	A função consegue alterar o grupo dono e o usuário dono do arquivo passado para ele, é chamado quando o comando de terminal *chown* é chamado

data de acesso e última data de modificação respectivamente. Além disso, houveram mudanças na função **getattr_chaosfs** para mostrar estas datas quando necessário, atualização da data de modificação no **utimens_chaosfs**, **read_chaosfs** e **write_chaosfs**.

2.3 Suporte aos proprietários dos arquivos

Primeiro foi adicionado na estrutura do *inode* mais dois itens, **gid** do tipo *gid_t* e **uid** do tipo *uid_t*, o qual representa a qual grupo pertence o arquivo e qual usuário pertence o arquivo, respectivamente

2.4 Aumento do limite de arquivos criados

Primeiro avaliamos a forma como os *inodes* dos arquivos eram guardados no superbloco. Assim, como o superbloco ocupava apenas um bloco do sistema, foi necessário realizar uma operação matemática avaliando quantos blocos seriam necessários, para que fosse possível armazenar uma quantidade suficiente de *inodes* para 1024 arquivos.

Após isso, foi necessário alterar as chamadas realizadas a função **preenche_bloco**, para que os blocos presentes no superbloco não fossem preenchidos com conteúdo dos arquivos.

3 Limites do sistema de arquivo e Escalabilidade

O ChaosFs se propõe a implementar algumas funções necessárias na utilização de um sistema de arquivos genérico. Entretanto, algumas funcionalidades não puderam ser implementadas, ou possuem limitações. A quantidade máxima de arquivos que podem ser criados pelo usuário é 1024, e cada arquivo só pode ocupar, no máximo, o equivalente a 1 bloco ou 4096 bytes. Remoção e criação de arquivos e diretórios é suportada assim como modificar o grupo e o dono do arquivo e seus direitos.

A performance de escrita do nosso FS é inferior a outros mais famosos, como ext4, por conta do seu comportamento de copiar todo o disco virtual, em memoria principal, para o disco em cada escrita. Por conta disso, utilizações que exigem uma boa resposta em diversas requisições de escrita, percebemos que nosso FS é insatisfatório. Além disso, não existe controle de quais arquivos já estão abertos e não podem ser sobrescritos.

O tamanho máximo de cada arquivo é de 4096 bytes, o que equivale a um bloco. Houve a tentativa de implementar uma versão que suportasse arquivos maiores, de até 64MB e que ocupassem mais de um bloco [1], até foi possível armazenar o conteúdo nos blocos de maneira encadeada, entretanto, houveram dificuldades durante a adaptação das funções de leitura e escrita do FUSE para ler os blocos de maneira encadeada.

O maior tamanho de disco suportado pelo FS é de, aproximadamente, 4.5Mb, o tamanho que todos os 1024 arquivos e seus inodes ocupam. Também existiu uma tentativa de implementar uma versão que pudesse receber como argumento o tamanho total desejado [2] , mas tivemos problemas em salvar os dados do disco virtual no arquivo do nosso FS. O método consegue criar um arquivo com o tamanho correto, carregar seus dados e efetuar as operações sem nenhum tipo de erro, mas os dados não são, de fato, armazenados. A ideia era manter o limite máximo de arquivos em 1024, mas ter um disco grande que pudesse armazenar arquivos grandes.

Apesar de conseguir criar e remover diretórios, não é possível criar arquivos e outros diretórios dentro dos mesmos e/ou listá-los, pois nos faltou a reimplementação da função *readdir* [3]. A solução escolhida foi que para cada diretório armazenar em seu bloco no disco os números dos *inodes* que estão dentro do dele, tivemos como dificuldade principal a parte de conseguir armazenar e ler os *inodes* dentro de um diretório.

Em decorrência das limitações do sistema não é possível classificar o ChaosFS como escalável.

4 Instruções para compilação e execução

A compilação do ChaosFS foi implementada utilizando um **Makefile**, logo, para compilar, basta o comando *make*. Através dele indicamos as *flags* ao compilador, a rotina para limpar os binários antigos e construir o *build*. Além das *flags* recomendadas no escopo do projeto,

[1] Versão contida no *branch* **tam_max_64**

[2] Esta versão se encontra no *branch* **disc_tam_arbitr**

[3] As tentativa de fazer esta função está no *branch* **diretorio**

adicionamos -std=gnu11 para garantir que não haja inconsistência nos laços.

Requirimos o uso do ***Fuse*** na versão 3.1 ou superior, não houve verificação em versões anteriores a esta. Por isso, não garantimos seu comportamento em versões anteriores.

Após a compilação, no diretório do projeto, pode-se executar com o seguinte comando:

```
./build/chaosfs -f -d -s <DIRETORIO>
```

em que **<DIRETORIO>** pode ser substituído o local que se deixa construir o sistema de arquivo. Desta forma o programa verificará se existe um arquivo chamado **/tmp/chaos.fs** no seu sistema de arquivos padrão. Neste arquivo que guardamos os dados da persistência. Se o arquivo existe, então o utilizaremos como fonte inicial de dados, caso não exista, é criado um arquivo em branco no tamanho do nosso *FileSystem*.

Em determinado momento, se for desejado formatar o disco e começar com um novo *FileSystem* em branco, sempre será possível rodar o ChaosFS com o argumento que o formata, "**-ffs**"(*format file system*).

```
./build/chaosfs -f -d -s /tmp/seu_nome -ffs
```

Adicionalmente, criamos um *script* em **Python** para verificar a capacidade de criação de no máximo 1024 arquivos de 4096 bytes. O programa cria uma quantidade arbitrária de arquivos todos com o tamanho máximo suportado para testar se o que fazemos condiz com o esperado. Para execução do programa basta executar:

```
python teste.py <quantidade> <diretorio do FS>
```

5 Trabalho Futuros

Para trabalhos futuros elencamos as seguintes tarefas que poderiam ser melhoradas na implementação. O processo de salva de arquivo não é eficiente, ou seja, a cada alteração salva-se todo o disco novamente, mesmo que não haja alteração no arquivo. Além disso, gostaríamos do suporte correto a diretório, tendo em vista uma limitação na exibição dos arquivos no disco.

Como atributos que gostaríamos de adicionar, indicamos o suporte a link e atalhos; a implementação de um processo de apagar seguro.