Universidade Federal do ABC

MCTA026-13

Sistemas Operacionais

Apocalypse FS

Autores:	*RA:*
Guilherme Melro Salim	11073612
Marcelo Martins Oliveira	11012215
Mauro Mascarenhas de Araújo	11020215

6 de Maio de 2019

Conteúdo

1 Introdução

Os sistemas de arquivos (SA),ou sistemas de ficheiros, é um sistema que gerencia os dados em algum meio de armazenamento de dados em massa, seja em uma mídia removível ou "fixa", como os HDDs (do inglês, *Hard Disk Drive*) [4]. Alguns dos sistemas mais conhecidos são o NTFS utilizado pelo Windows, o FAT/FAT32 normalmente utilizado em mídias removíveis e o EXT(2/3/4) utilizado por sistemas UNIX.

Muitas vezes não percebemos que o sistema operacional facilita muito o trabalho do programador e, consequentemente, a vida do usuário, ao disponibilizar APIs de "alto nível"que tornam as requisições de acesso à recursos do sistema mais transparentes. Um dos responsáveis por prover esta interface é o sistema de arquivos, que controla como os dados são armazenados e recuperados, sem o usuário ou o programa necessitar saber onde está o arquivo (ou as partes) dele no sistema [4].

A fim de entender melhor como funcionam os sistemas de arquivos, foi proposta uma atividade prática onde devemos elaborar nosso próprio sistema de arquivos, de acordo com o roteiro estabelecido pelo Prof. Dr. Emílio Francesquini [1].

2 Sistema base (BrisaFS)

A fim de exemplificar o funcionamento de um sistema de arquivos e prover uma base para o desenvolvimento da atividade, foi disponibilizado um sistema básico chamado "BrisaFS"que, ao ser encerrado, perde toda a informação processada.

2.1 Análise do caso

O sistema proposto utiliza o **FUSE** para fazer o intermédio entre o Kernel do Linux e o sistema de arquivos desenvolvido (como em uma SandBox)[3].

A atividade proposta consiste no cumprimento de algumas *milestones*, que encontram-se descritas na página do projeto[1].

2.1.1 Limitações

Justamente por ser um sistema básico, ele conta com diversas limitações que devem ser implementadas no decorrer do cumprimento das *milestones*. Algumas delas são:

- Falta de persistência de arquivos.
- Falta de suporte à diretórios.
- Falta de suporte à definição de horários.
- A quantidade máxima de arquivos é muito limitada.
- O tamanho máximo do arquivo é restrito à 4096 bytes.
- Dentre outras limitações.

3 Implementação

O Apocalypse FS foi implementado diretamente sobre o BrisaFS, portanto, ele compartilha de suas características base e, consequentemente, de algumas de suas limitações.

Atualmente o sistema está limitado a um máximo de 65536 arquivos e diretórios somados. Este limite está imposto pela macro **MAX_FILES** que pode ser modificada para uma quantidade maior ou menor, dependendo da quantidade de RAM que o usuário dispõe.

Cada bloco está estritamente ligado à um *inode* que, caso não esteja vazio, possui a propriedade "**block**» 0. Isso facilita no momendo de remover arquivos e diretórios, uma vez que evita operações de escrita desnecessárias, já que basta definir a propriedade como "0"no *inode* para que o bloco seja considerado vazio.

3.1 Cumprimento de *milestones*

Todas as *milestones* equivalentes à nota 7 foram cumpridas, além de a implementação já conter base para o suporte à discos de tamanhos arbitrários e o tamanho máximo dos arquivos já são arbitrários, ou seja, suporta arquivos de 1GB+.

Cada diretório assume um *inode* tal qual um bloco de dados que é particionado da forma **BLOCK_SIZE/sizeof(unsigned int)**, a fim de que possa guardar o índice para os inodes dos outros diretórios e arquivos contidos nele.

Por fim, para que o sistema possa lidar com arquivos de tamanhos arbitrários, ele é construído sobre uma estrutura similar à uma lista ligada, onde o *inode* principal armazena os dados do arquivo e os demais servem para apontar para a localização

do próximo. Isso só é possível, pois, na implementação atual, cada *inode* está expressamente atrelado à um bloco de dados. Logo, cada arquivo de tamanho X (bytes) ocupa **X/BLOCK_SIZE** blocos (e seus respectivos *inodes*), onde **BLOCK_SIZE** está atualmente definido como 4096 bytes.

3.1.1 Resolução de problemas

Bem como o sistema base, a principal estrutura envolvida foi o vetor (ou *array*, em inglês), tanto para o armazenamento dos *inodes*, quanto para o armazenamento das sequências de blocos, que contêm os dados úteis.

Com excessão da operação de escaneamento de diretório, onde o encadeamento de algoritmos resulta na execução de $O(n^2)$ verificações nos inodes, todas as demais operações possuem tempo de execução O(n).

3.2 Recursos adicionais

Além das milestones cumpridas, foi implementada a função de renomeação de arquivos, onde o usuário pode trocar o arquivo de nome ou de diretório sem que haja modificação drástica no *inode* do arquivo em questão e todas as referências em blocos de diretórios são atualizadas.

3.3 Compilando e executando

A fim de facilitar a compilação e execução do programa, basta instalar as dependências como descritas no arquivo "**README.md**"disponível na raiz do repositório do projeto[2].

Após instalar as dependências e clonar o repositório, navegue até o diretório "*./src/*"e execute as seguintes linhas no terminal (As linhas que começam com '#' devem ser ignoradas, pois, trata-se apenas de comentários):

```
# Limpa todos os arquivos ja compilados
make clean
# Compila o codigo fonte disponivel no repositorio
make
# Executa o sistema de arquivos no caminho padrao
make run
```

Atenção : caso seja necessário alterar o ponto de montagem, é possível especificar no parâmetro da linha de comando "*make run*". Note que «diretorio¿"é o diretório do ponto de montagem do sistema de arquivos:

```
# Executa o sistema de arquivos no caminho especificado
make run MOUNT=<diretorio>
```

3.4 Expansões

Além das *milestones* não atingidas, é possível melhorar o desempenho do programa (i.e., fazer o uso de algoritmos mais eficientes, tanto com relação ao consumo de espaço, quanto em relação ao tempo de execução).

Por fim, caso o projeto alcance um bom nível de maturidade, é possível implementá-lo diretamente no kernel do Linux, ou seja, é possível compilá-lo como parte do "cérebro"do sistema, sem o auxílio de uma "SandBox".

4 Conclusão

Através da elaboração deste projeto, pode-se concluir que, nem sempre as soluções encontradas em alto nível são realmente triviais como parecem ser. Vários subsistemas são responsáveis por manter a transparência do funcionamento do SO para o usuário final.

Com isso, pode-se concluir que, o desempenho e a consistência de um sistema operacional não depende somente da implementação de seu kernel, mas de seus subsistemas, como o de arquivos.

Ao tratar especificamente do sistema de arquivos é possível compreender a importância do uso adequado de estruturas de dados e de implementar os algoritmos de forma adequada, a fim de que o usuário final possa tirar o melhor proveito possível do sistema (um dos exemplos do que não deve ser feito, é o fato de o bloco inteiro de memória estar sendo salvo quando a função "**fsync**"é chamada).

Referências

[1] Emílio Francesquini. Projeto de programação. http://professor.ufabc.edu.br/~e.francesquini/2019.q1.so/projeto/, 2019. Online. Acessado em 05 de maio de 2019.

[2] Salim, Guilherme and Martins, Marcelo and Mascarenhas, Mauro. Apocalypsefs. https://github.com/ufabc-bcc/2019_Q1_SO_BrisaFS-ApocalypseFS/, 2019. Online. Acessado em 05 de maio de 2019.

[3] Wikipedia contributors. Filesystem in userspace — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Filesystem_in_Userspace&oldid=892362371, 2019. Online. Acessado em 05 de maio de 2019.

[4] Wikipédia. Sistema de ficheiros — wikipédia, a enciclopédia livre. https://pt.wikipedia.org/w/index.php?title=Sistema_de_ficheiros&oldid=54485214, 2019. Online. Acessado em 05 de maio de 2019.