

Escola Politécnica da Universidade de São Paulo
PCS3732 - Laboratório de Processadores



Ivan Mathias Sant'anna e Silva - 10773586
Rodrigo Leal Reis Morello - 13695827
Vitor Thompson Borges - 13682941

SFS - Simple File System

São Paulo
2025

Sumário

1. Introdução.....	3
2. Base Teórica.....	4
3. Premissas do Projeto.....	5
4. Implementação.....	7
4.1. Arquitetura.....	7
4.2. Estrutura Básica para Bare-Metal.....	10
4.3. Código.....	12
4.3.1. include.....	13
4.3.2. src/core.....	14
4.3.3. src/system.....	16
5. Testes.....	20
6. Considerações Finais.....	24

1. Introdução

Como projeto final para a disciplina PCS3732 - Laboratório de Processadores, ministrada pelos professores Bruno Basseto e Carlos Cugnasca, o grupo escolheu desenvolver um sistema de arquivos simplificado, assim surgiu o SimpleFileSystem - SFS.

Um sistema de arquivos pode ser definido como sendo a estrutura lógica que um sistema operacional utiliza para organizar, armazenar e gerenciar arquivos em um dispositivo de armazenamento. Esse programa é responsável por diversas funções importantes no contexto de gerenciamento de arquivos, como por exemplo, gerencia espaço para alocar blocos de arquivos, gerencia os metadados que são informações (nome, tamanho, data de criação, permissões, etc.) que descrevem os arquivos, controla o acesso e segurança por meio de permissões atribuídas a usuário e arquivo, e fornece uma interface para que outros programas possam criar, editar, ler e excluir arquivos diretamente.

Dessa forma, o grupo optou por implementar um sistema de arquivos simplificado que possibilitasse que o usuário, por meio de um terminal, pudesse criar, ler, editar e excluir apenas arquivos texto (*.txt*), e pudesse criar e navegar entre diretórios, além de poder excluir um diretório vazio. Inicialmente, a proposta do grupo era seguir a estrutura de sistema de arquivos FAT32, que já foi utilizada em sistemas Windows, contudo, após pesquisas mais aprofundadas, o grupo decidiu mudar e implementar um sistema de arquivos baseado na estrutura inode, utilizada em sistemas da família Unix.

2. Base Teórica

O projeto visa desenvolver um sistema de arquivos baseado na estrutura inode, por isso, é de extrema importância entender melhor os conceitos e as características dessa estrutura antes de implementá-la.

O inode, ou index node (nó de índice), é uma estrutura de dados fundamental para o armazenamento de metadados de arquivos. Entre os dados que um inode armazena estão o tipo do arquivo, permissões, tamanho do arquivo, carimbos de tempo e data, e ponteiros para os blocos de dados. Esses ponteiros podem apontar diretamente para os blocos de dados que contenham o conteúdo do arquivo, ou ainda, podem apontar para um outro bloco de ponteiros que, estes sim, apontam para blocos de dados, chamados agora de bloco de dados indireto, ou ainda, esses últimos ponteiros podem apontar para outro bloco de ponteiros que irão apontar para blocos de dados chamados de bloco de dados duplamente indiretos, e assim sucessivamente. Essa estrutura de ponteiros para blocos de dados (diretos e indiretos) permite o gerenciamento de arquivos maiores e também permite a escalabilidade de um arquivo.

Um diretório é considerado um tipo especial de arquivo, cujo conteúdo consiste em uma lista de pares contendo um nome de arquivo e o número do inode correspondente. Quando um sistema de arquivos é criado ou formatado, uma tabela de inodes é alocada em um local fixo no disco, e o número de inodes possíveis é predefinido e fixo em muitos sistemas. Assim, o inode é a base para a organização e o gerenciamento de arquivos em um sistema.

Para ilustrar o processo de acesso a um arquivo, consideremos o comando no terminal Linux `/home/usuario/documento.txt`. Primeiramente, o sistema de arquivos acessa o inode do diretório raiz `/` para localizar o inode de `home`. Em seguida, dentro do inode de `home`, o processo se repete para encontrar o inode do diretório `usuario`. Dentro do inode de `usuario`, o sistema busca pelo arquivo `documento.txt`. Uma vez encontrado, o sistema lê o inode correspondente ao arquivo e, por meio dos ponteiros de bloco de dados, consegue localizar e ler o conteúdo do arquivo diretamente do disco.

A imagem abaixo exemplifica a estrutura do inode e seus ponteiros diretos e indiretos para bloco de dados.

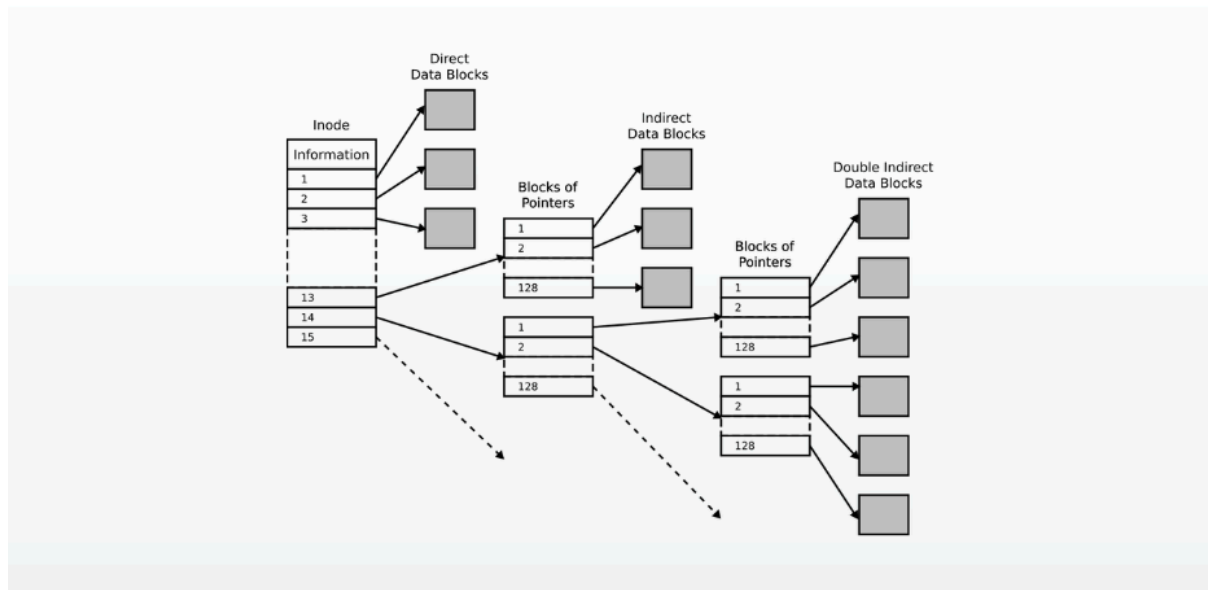


Figura 1: Estrutura geral do inode

3. Premissas do Projeto

Para criar um sistema de arquivos, o grupo adotou algumas premissas para o seu desenvolvimento. Dentre elas, adotou-se uma implementação bare-metal, ou seja, uma implementação que possibilita interagir diretamente com o hardware sem o intermédio de um sistema operacional, dessa forma, o sistema de arquivos poderia rodar na Raspberry Pi como sendo um programa de usuário.

Além disso, o grupo optou por utilizar a própria memória RAM da Raspberry Pi como disco para armazenamento de arquivos e diretórios. É crucial notar que, por ser uma memória volátil, todos os dados serão perdidos ao desligar a Raspberry Pi. Embora o cartão SD, que contém a imagem do programa, pudesse ser usado como disco não volátil, isso exigiria a criação de um driver para comunicação direta com a alocação de memória do cartão, além de configurá-lo para suportar a estrutura i-node.

Vale destacar que a Raspberry Pi 3, modelo utilizado pelo grupo, possui 1 GB de memória RAM, sendo mais do que o suficiente para a execução do nosso

programa, uma vez que utilizaremos apenas uma parte dela para o armazenamento dos diretórios e dos arquivos. Isso pode ser visto como uma limitação num primeiro momento, mas como a ideia principal do projeto é a lógica do sistema de arquivos, a sua demonstração pode ser feita utilizando arquivos de texto simples que não necessitem de tanta memória para armazenamento.

O projeto tem como ideia possibilitar que o usuário, a partir de um terminal, execute os seguintes comandos básicos inicialmente:

- *touch*: criar um arquivo txt;
- *mkdir*: criar um diretório;
- *ls*: listar o conteúdo de um diretório;
- *cd*: navegar entre diretórios;
- *cat*: leitura do conteúdo de um arquivo;
- *write*: escrever um conteúdo texto em um arquivo;
- *rm*: excluir arquivos e diretórios vazios;
- *stat*: visualizar estatísticas de utilização da memória.

Para a realização do projeto, o grupo optou pela seguinte implementação:

- Códigos escritos na linguagem C;
- Raspberry Pi 3 (própria de um dos integrantes);
- Arquitetura alvo de 32 bits;
- Comunicação serial via Mini UART;

Vale ressaltar ainda que apesar da Raspberry Pi 3 apresentar suporte a arquitetura de 64 bits do processador ARM, o grupo optou por seguir a arquitetura de 32 bits, uma vez que essa foi trabalhada ao longo das aulas da disciplina.

4. Implementação

Com a base teórica e as premissas do projeto definidas, é possível partir para a implementação de fato do SFS. Assim, abaixo estão descritos a arquitetura, a estrutura básica e as funções implementadas dentro do sistema de arquivos.

4.1. Arquitetura

A partir da ideia do projeto, estabeleceu-se a arquitetura que seria implementada para o sistema de arquivos SFS. A definição da arquitetura é de extrema importância para garantir o desempenho e integridade do sistema.

Inspirando-se em sistemas de arquivos tradicionais como o *ext4*, que utiliza a estrutura de inodes para gerenciar arquivos e diretórios, a equipe projetou uma estrutura lógica dividida em blocos bem definidos. A imagem a seguir ilustra de forma conceitual a organização de um sistema de arquivos baseado em inodes, similar à base adotada para o SFS.

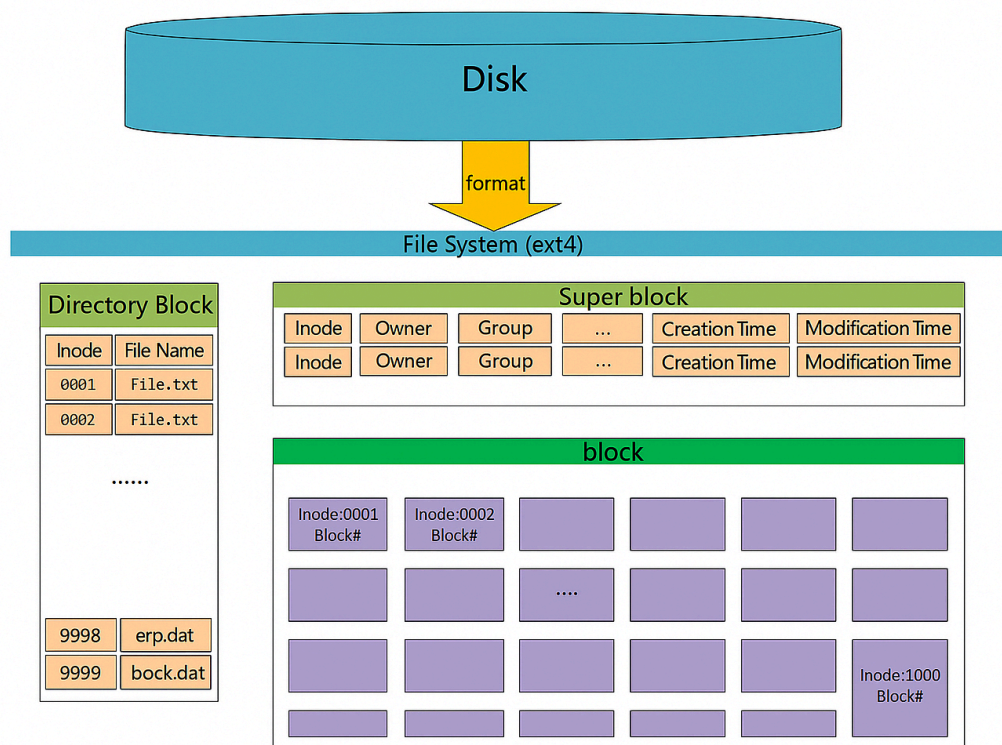


Figura 2: Exemplo de arquitetura baseada em inode

Na estrutura dividida em blocos utilizada pelo SFS, cada um deles apresenta funções e características específicas, salientando que os blocos apresentam algumas simplificações quando comparados com sistemas mais complexos, como o *ext4* citado anteriormente. Dessa forma, segue abaixo a listagem dos blocos implementados no projeto.

Superbloco: Consiste no primeiro bloco (bloco 0) e é o ponto de partida de acesso ao disco, pois contém metadados essenciais que definem a estrutura e o estado geral do disco. As informações armazenadas nesse bloco são:

- **Magic Number (Número Mágico):** um valor único que identifica o sistema de arquivos, a fim de prevenir montagens incorretas do sistema ou acesso a dados corrompidos.
- **Número Total de Blocos:** define o tamanho total do disco virtual em termos de blocos.
- **Endereços de Início das Seções de Metadados:** armazena ponteiros para os endereços iniciais do Bitmap de Inodes, Bitmap dos Blocos de Dados, Tabela de Inodes e Área de Dados.

Bitmap de Inodes: É um mapa de bits que rastreia os status de cada inode da Tabela de Inodes, ou seja, mapeia se cada inode está em uso ou livre.

- Quando um novo arquivo é criado, o sistema de arquivos consulta o bitmap para encontrar o primeiro bit livre, aloca o inode correspondente e marca o bit como usado.
- Quando um arquivo é excluído, o bit do inode correspondente é liberado.

Tabela de Inodes: Representa uma área contígua que armazena um array de todas as estruturas inode do sistema. Um inode não armazena o conteúdo de um arquivo ou diretório, mas sim informações que os descrevem. No caso do nosso sistema de arquivos simplificado, o inode armazenará as seguintes informações:

- Tipo: identifica se é um arquivo normal ou diretório;
- Tamanho do arquivo;
- Ponteiros para os blocos de dados.

Bitmap de Bloco de Dados: Consiste em um mapa de bits que rastreia o uso dos blocos na Área de Dados, sendo fundamental para o gerenciamento dessa estrutura. Ele monitora:

- Quais blocos na Área de Dados estão ocupados com conteúdo de arquivos ou diretórios;
- Quais blocos estão livres para novas alocações.

Área de Dados: Consiste no restante do espaço do disco, após as seções de metadados. Nela, o conteúdo real dos arquivos e as listas de entradas de diretório são armazenados.

- O conteúdo dos arquivos contém os dados criados pelo usuário.
- Os diretórios, considerados arquivos especiais, contêm listas de entradas com pares que associam um nome de arquivo/diretório ao seu número de inode correspondente (como descrito no post anterior).

Aqui vale uma observação quanto a uma simplificação adotada pelo grupo especialmente quanto à estrutura do inode descrita na Tabela de Inodes. O inode descrito pelo grupo agrupa apenas três metadados sobre os arquivos, enquanto que em um sistema, como o *ext4*, o inode armazena muito mais informações. Assim, pode-se construir uma tabela mostrando todos os metadados armazenados no sistema *ext4* e com a indicação de quais estão presentes no SFS.

Metadado	<i>ext4</i>	SFS
Tipo do arquivo	✓	✓
Permissões	✓	✗
<i>Owner</i>	✓	✗
Tamanho do arquivo	✓	✓
Data e horário importantes	✓	✗
<i>Link count</i>	✓	✗
Ponteiros de bloco de dados	✓*	✓*

Tabela 1: Comparação entre os metadados armazenados pelo inode no *ext4* e no SFS

Ainda na comparação, é preciso destacar que apesar de ambos terem ponteiros para blocos de dados, no caso do SFS foi implementado apenas ponteiros para blocos de dados diretos, pois como a ideia é implementar um sistema de arquivos simplificado, logo, os arquivos de texto a ser criado também serão simples, portanto não precisarão dos blocos de dados indiretos. No caso do *ext4*, os ponteiros podem ser indiretos, duplamente indiretos e até triplamente indiretos.

4.2. Estrutura Básica para *Bare-Metal*

Antes de implementar de fato o SFS, foi preciso desenvolver primeiro a base mínima de software necessária para que o sistema de arquivos pudesse funcionar diretamente no hardware da Raspberry Pi 3, sem depender de um sistema operacional. Isso é o que chamamos de ambiente *bare-metal*, ou seja, um ambiente onde temos controle total do hardware desde o primeiro ciclo de clock, com todos os recursos de software construídos manualmente.

Para isso, implementou-se três componentes essenciais, que trabalham de forma integrada.

Inicialização do Sistema (*Boot*): A primeira função de qualquer sistema bare-metal é assumir o controle total do hardware logo após o boot. Esse processo de inicialização prepara a memória, configura a pilha e define o ponto de entrada para o restante do código escrito em C.

Além disso, o sistema é programado para manter o processador em segurança caso ocorra algum erro inesperado, impedindo que instruções incorretas sejam executadas. Isso é essencial para garantir a estabilidade do ambiente.

No nosso projeto, o boot será realizado pelo código contido em um arquivo chamado *startup.s*, que será responsável por realizar todas as tarefas acima.

Organização da Memória: Como não há um sistema operacional gerenciando a memória, foi necessário definir exatamente onde e como o programa será carregado na RAM da Raspberry Pi.

Isso inclui:

- Definir onde o programa começa a ser executado na memória.
- Garantir que o código de inicialização seja o primeiro a rodar.
- Reservar regiões específicas para variáveis, funções e buffers.

Essa organização permite que o sistema funcione de forma previsível, evitando conflitos ou sobreposição de dados na memória. Isso será feito pelo *linker script* (*linker.ld*), que consiste em um arquivo de configuração que instrui o *linker* a organizar o nosso programa em um layout de memória específico.

Comunicação Serial com o Usuário: Para interagir com o sistema (exibir mensagens, depurar o funcionamento ou receber comandos), foi implementado um driver de comunicação via Mini UART, que é a porta serial da Raspberry Pi 3.

Essa interface é responsável por:

- Configurar os pinos físicos da Raspberry Pi para envio e recebimento de dados.
- Estabelecer a velocidade da comunicação (baud rate).
- Implementar funções de envio e leitura de caracteres em tempo real.

Essa funcionalidade é o canal de comunicação principal com o sistema bare-metal, e será essencial para a próxima fase, quando o sistema de arquivos começar a ser testado e controlado. No nosso projeto, o driver está descrito no arquivo *uart.c* e possui as seguintes funções e configurações:

- Função de Inicialização (*uart_init*): Nesta função,
 - os pinos 14 e 15 da GPIO são reconfigurados de "entrada/saída de propósito geral" para sua função alternativa de comunicação serial;
 - e o periférico da Mini UART é habilitado e configurado para operar no modo de 8 bits de dados e com um baud rate de 115200.
- Funções de Envio e Recebimento (*uart_putc* e *uart_getc*): Estas funções implementam a comunicação através de "polling". Elas ficam em um loop, lendo continuamente um registrador de status até que um bit específico indique que o hardware está pronto para enviar um novo caractere ou que um caractere foi recebido.

Juntos, esses três elementos formam a fundação do sistema bare-metal. A inicialização garante um ponto de partida seguro, a organização da memória define um espaço de trabalho bem estruturado, e a comunicação serial permite interações externas com o sistema..

Vale ressaltar ainda que o grupo utilizou como referência para essa etapa do projeto, o seguinte repositório no [github](#), em que o autor descreve o processo de construir um ambiente bare-metal na Raspberry Pi 3, incluindo a comunicação com a mini UART nesse módulo referenciado.

4.3. Código

A implementação do código consiste na parte funcional do sistema de arquivos de fato. Nessa etapa, vamos descrever com foco no desenvolvimento das estruturas de dados e das funções de sistema. Para entender melhor, é possível passar por cada arquivo do programa, sendo que a estrutura do código que se encontra no [repositório do github do grupo](#) encontra-se abaixo.

- */include*
 - *common.h*
 - *fs_defs.h*
 - *sfs.h*
 - *shell.h*
 - *uart.h*
- */src*
 - *core*
 - *common.c*
 - *kernel.c*
 - *shell.c*
 - *uart.c*
 - *system*
 - *sfs.c*
 - *dir.c*
 - *file.c*

Vale ressaltar que tem outros arquivos no repositório mas que consistem em nos arquivos já descritos anteriormente, como *linker.ld*, e outros arquivos que são necessários naturalmente, como o *Makefile*.

4.3.1. *include*

Os arquivos *.h* que se encontram dentro da pasta *include* consistem basicamente na declaração das funções principais e secundárias que serão usadas pelo SFS, de forma que estão separadas dessa forma para funcionarem como bibliotecas, além de organizar e facilitar a utilização dessas funções nos códigos em C.

Como a implementação das funções declaradas aqui estão apenas nos códigos em C, a explicação delas será feita nos tópicos seguintes do relatório para evitar repetição. Entretanto, o arquivo *fs_defs.h* merece um destaque importante aqui, pois além de ter declaração de certas funções, é nesse arquivo que foram implementadas as configurações do sistema de arquivos e as estruturas dos blocos principais.

Constantes de Configuração: Foram definidas constantes para os parâmetros do sistema, descritos e definidos abaixo.

- *BLOCK_SIZE* define o tamanho de cada bloco de dados e, no nosso caso, consiste em 512 bytes.
- *MAX_FILENAME_LEN* define o comprimento máximo para nomes de arquivos e diretórios como 28 caracteres.
- *NUM_INODES* limita o sistema a um máximo de 128 arquivos e diretórios.
- *NUM_DATA_BLOCKS* define o tamanho total do nosso disco em 8192 blocos, o que resulta em um disco de 4MB, por causa dos blocos de 512 bytes.
- *MAX_DIRECT_POINTERS* define que cada inode pode apontar diretamente para um máximo de 12 blocos de dados.
- *FS_MAGIC* é um "número mágico" único que é escrito no superbloco para identificar o disco como sendo formatado pelo nosso sistema de arquivos.

- *ATTR_FILE* 1 e *ATTR_DIRECTORY* 2 são as constantes usadas no campo *tipo* do inode para diferenciar se o objeto é um arquivo ou um diretório.

Estruturas dos Blocos Principais: Dentre as estruturas descritas anteriormente, aqui implementaram-se, usando o struct, as três estruturas principais de dados.

- **Superblock** (Superbloco)
 - Armazena os metadados globais do sistema;
 - Armazena o número mágico e os endereços de início do bitmap de inodes, bitmap de dados, tabela de inodes e área de dados.
- **Inode**
 - Contém os metadados de um arquivo ou diretório, sendo eles
 - *type* (tipo): definidos nas constantes (arquivo ou diretório),
 - *size*: tamanho do arquivo em bytes;
 - *direct_pointers*: um array de ponteiros diretos que apontam para os blocos de dados que contêm o conteúdo.
- **DirectoryEntry** (Entrada do Diretório)
 - Estrutura para entrada dos diretórios;
 - Consiste em um par nome do arquivo (*filename*) e o número do inode (*inode_number*) correspondente a ele.

4.3.2. *src/core*

Dentro desta pasta, os arquivos implementam funções majoritariamente auxiliares para o funcionamento do sistema de arquivos, isto é, não implementam de fato os comandos de terminal que o usuário pode usar.

common.c

Implementa funções, que normalmente são fornecidas por bibliotecas padrões, responsáveis por realizar manipulação de strings e memória.

- *strlen*: calcula o comprimento de uma string, contando o número de caracteres até encontrar o caractere de terminação “\0”.
- *strcmp*: compara duas strings e retorna 0 se forem iguais.
- *strcpy*: copia o conteúdo de uma string de origem para um string de destino.
- *memcpy*: copia um número *n* de bytes de uma região de memória de origem para uma de destino, pode copiar qualquer tipo de dado.
- *memset*: preenche os primeiros *n* bytes de uma região de memória com um valor constante.
- *strcat*: concatena uma string de origem ao final de uma string de destino.
- *itoa*: converte um inteiro para uma string.

kernel.c

Representa o ponto de entrada principal para a parte do programa escrita em linguagem C. Sua função *main()* é a primeira rotina C a ser executada, sendo chamada diretamente pelo código de inicialização em Assembly (*startup.s*).

Esse arquivo atua como o orquestrador do sistema, pois ele não contém a lógica detalhada do sistema de arquivos, mas é responsável por inicializar todos os módulos principais na sequência correta e, por fim, entregar o controle do programa ao shell interativo, que aguardará os comandos do usuário.

Assim, primeiramente, a função *main* inicializa a comunicação serial, imprime no terminal algumas linhas de formatação, chama as funções que realizarão a formatação e a montagem do SFS, e por último, chama a função *shell_start*.

shell.c

Junto a transmissão serial via mini UART, foi implementado o arquivo *shell.c* de forma a complementar a transmissão. As funções desse arquivo são responsáveis por, continuamente, aguardar, ler o comando do terminal digitado pelo usuário, identificar o comando e chamar a função correta a correspondente ao comando, ou seja, basicamente faz a tradução das entradas do usuário em operações concretas dentro do sistema de arquivos.

A implementação foi dividida em um loop principal que gerencia a interação com o usuário e duas funções auxiliares que lidam com a leitura e a interpretação dos comandos.

Loop Principal - *shell_start()*: A função *shell_start* opera em loop infinito (usando um *while(1)*), com o objetivo de manter o sistema ativo continuamente e preparado para receber os comandos do usuário. Assim, em cada iteração do loop, a função executa um ciclo que consiste nas seguintes etapas:

- Exibição do diretório de trabalho atual, que mostra na tela algo como *SimpleFS:/\$* ou *SimpleFS:<diretorio>\$*;
- Leitura da entrada do usuário com ajuda da função auxiliar *read_command* que aguarda e captura via UART a linha de texto inteira digitada pelo usuário;
- Análise do comando com a função auxiliar *parse_command* que separa a entrada do usuário em comando e seus argumentos;
- Por fim, é feita a leitura do comando de terminal solicitado e então é feita a chamada da função que implementa tal comando, passando também os argumentos quando necessários.

uart.c

Implementa as funções necessária para a implementação da transmissão serial via Mini UART descrita em tópicos anteriores.

4.3.3. *src/system*

Aqui os arquivos implementam as funções correspondentes aos comandos de terminal que torna o sistema de arquivos de fato funcional.

sfs.c

O arquivo *core.c* é responsável por conter funções auxiliares importantes que atuarão diretamente ligadas às funções de terminal do sistema de arquivos, além de ter as funções de formatação e montagem do sistema.

- *read_block*: realiza a leitura de um bloco do disco.
- *write_block*: realiza a escrita de um bloco no disco.
- *set_bitmap_bit*: marca um bit individualmente do bitmap como ocupado.

- *clear_bitmap_bit*: marca um bit individualmente do bitmap como livre.
- *find_free_inode*: percorre o bitmap de inodes para encontrar o primeiro inode livre.
- *find_free_data_block*: percorre o bitmap de bloco de dados para encontrar o primeiro bloco de dados livre.
- *find_entry*: procura um arquivo ou um diretório pelo nome dentro do diretório atual.
- *fs_get_current_path*: retorna o caminho do diretório de trabalho atual.
- *fs_format*: responsável por criar no disco uma estrutura de sistema de arquivos vazia e válida. Assim nessa função, a lógica implementada consiste em:
 - Criação e configuração do Superbloco e sua escrita na memória no primeiro bloco do disco (definido como *ram_disk*);
 - As áreas que serão usadas para os bitmaps e a tabela de inodes são zeradas para garantir que não contenham lixo de memória;
 - Realiza a chamada para a função de montagem *fs_mount*, a qual irá configurar os ponteiros globais;
 - Reservar os blocos de dados, que estão sendo usados pelos metadados, no bitmap de blocos de dados;
 - Cria o diretório raiz (/), o que envolve a alocação de um inode livre e a alocação de um bloco de dados, bem como configurar as entradas . e .. dentro desse diretório.
- *fs_mount*: realiza a preparação do sistema de arquivos para ser usado, assim realiza três etapas fundamentais, sendo elas
 - Validação do Superbloco, em que a função lê o bloco 0 do disco e verifica o *magic_number*, pois se o número não for o esperado, significa que disco não está formatado, sendo preciso chamar a função *fs_format* novamente;
 - Configuração dos Ponteiros Globais, sendo o passo mais importante da montagem, pois aqui é onde a função usa os endereços de início dos bitmaps e tabelas, definidos no superbloco, para configurar os ponteiros (*inode_bitmap*, *data_bitmap*, *inode_table*) que apontam diretamente para as localizações corretas dentro do disco;

- Define o diretório de trabalho atual como sendo o diretório raiz (/), localizado no inode 0.

dir.c

Implementa as funções correspondentes aos comandos de terminal que estão relacionados diretamente com manipulação de diretório, além da função que mostra as estatísticas de memória do SFS.

- *fs_mkdir*: tem o objetivo de criar um novo diretório, assim, segue uma lógica que consiste em verificar se o nome escolhido já existe, em seguida encontrar um inode e um bloco de dados livres usando as funções auxiliares, encontrar um espaço vazio no diretório pai para adicionar a nova entrada e, por fim, atualizar os bitmaps, a entrada do diretório pai e a tabela de inodes para refletir a criação do diretório.
- *fs_cd*: altera o diretório de trabalho atual. Assim, a lógica implementada consiste em buscar o diretório de destino solicitado e, caso ele exista, a variável global que armazena o inode correspondente ao diretório atual é atualizada com o inode do novo diretório atual. Além disso, também é necessário atualizar a variável que criamos para mostrar no terminal o diretório em que a pessoa se encontra no momento.
- *fs_ls*: tem o objetivo de listar os conteúdos, tanto arquivos quanto diretórios, presentes no diretório atual. Dessa forma, a lógica escolhida para a função *fs_ls* consiste em ler o inode do diretório atual, percorrer seus ponteiros diretos (*direct_pointers*), ler os blocos de dados apontados por eles e, por fim, imprimir no terminal as entradas válidas que constam dentro do diretório. Um detalhe interessante, é que caso a entrada seja um arquivo, ela é impressa com um hífen “-” na frente, mas caso seja um diretório, no caso seria um subdiretório, a saída é impressa com um “d” na frente.
- *fs_stat*: utilizado para mostrar as estatísticas de uso de memória do sistema de arquivos. Dessa forma, é um comando que percorre o bitmap de inodes e bitmap de bloco de dados contando quanto de cada está ocupado e, após isso, mostra no terminal a quantidade de inodes utilizados, a quantidade de blocos de dados usados, o espaço da memória utilizado pelos blocos em

bytes e o espaço útil total da memória em bytes (espaço destinado ao usuário, desconsidera os blocos de metadados).

file.c

Implementa as funções correspondentes aos comandos de terminal que estão relacionados diretamente com manipulação de arquivos, com exceção da função de excluir (*fs_rm*) que também pode ser aplicada a um diretório vazio.

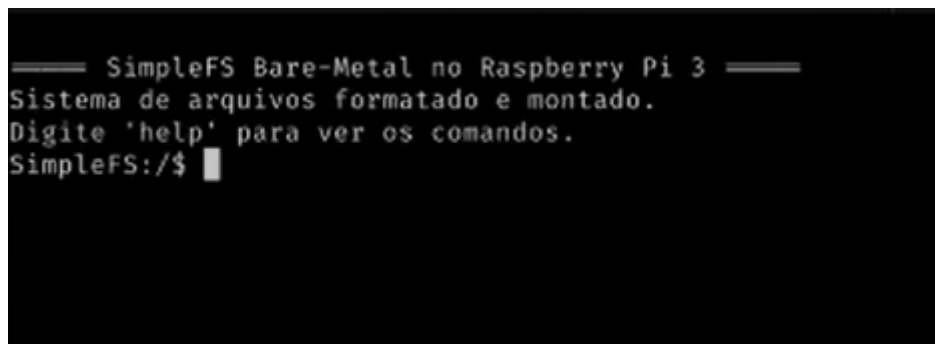
- *fs_touch*: tem o objetivo de criar um novo arquivo vazio (sem conteúdo) e a lógica implementada para esse comando segue a mesma do comando *mkdir*, uma vez que o processo de criação de arquivo e diretório é praticamente o mesmo, o que irá diferenciar de fato é o tipo de arquivo sendo criado.
- *fs_write*: serve para adicionar texto a um arquivo, e sua chamada é feita da seguinte forma: *write <nome_arquivo> <texto>*. A lógica implementada nessa função consiste em localizar o final do arquivo solicitado, preencher o espaço restante no último bloco. Caso seja necessário mais espaço, então é feita a alocação de novos blocos de dados, juntamente com a atualização do bitmap de bloco de dados e do inode do arquivo, adicionando mais um ponteiro direto.
- *fs_cat*: lê e exibe o conteúdo de um arquivo, assim, a lógica implementada consiste em encontrar o inode correspondente ao arquivo solicitado, percorrer a lista de ponteiros diretos para acessar os blocos de dados do arquivo e imprimir o conteúdo dos blocos no terminal.
- *fs_rm*: utilizado para excluir um arquivo ou excluir um diretório vazio. A lógica implementada nessa função consiste liberar os blocos de dados do item a ser excluído no bitmap de bloco de dados, liberar o inode do item no bitmap de inodes e, por fim, atualiza a tabela de entrada de diretório do diretório pai para remover o item.

5. Testes

Com toda a implementação necessária para SimpleFileSystem, foi possível realizar os testes de todos os comandos para demonstrar o funcionamento do projeto.

Para realizar os testes, foi preciso primeiro compilar e gerar o arquivo imagem. Para isso, basta rodar o comando *make* num terminal linux, para executar o arquivo *Makefile* que irá compilar e gerar códigos intermediários até gerar o arquivo de saída *kernel.img*. Em seguida, precisou copiar para o cartão SD os arquivos: *kernel.img*, *config.txt*, *bootcode.bin* e *start.elf*. Após isso, pôde-se colocar o cartão SD na Raspberry Pi 3, conectar o cabo serial nos pinos correspondentes, rodar o seguinte comando no terminal *screen /dev/tty.usbserial-XXXXX 115200* e, por fim, energizar a placa raspberry para testar o programa.

Ao iniciar o programa, a seguinte tela aparecerá no terminal.



```
==== SimpleFS Bare-Metal no Raspberry Pi 3 ====
Sistema de arquivos formatado e montado.
Digite 'help' para ver os comandos.
SimpleFS:/$
```

E ao digitar o comando *help*, todos os comandos possíveis de serem executados são listados na tela junto com a sua formatação de execução e uma breve explicação do que faz.

```

===== SimpleFS Bare-Metal no Raspberry Pi 3 =====
Sistema de arquivos formatado e montado.
Digite 'help' para ver os comandos.
SimpleFS:/$ help
Comandos disponiveis:
  ls                - Lista arquivos e diretorios
  mkdir <n>         - Cria um novo diretorio
  touch <n>         - Cria um novo arquivo vazio
  cat <n>           - Mostra o conteudo de um arquivo
  write <f> <t>     - Escreve/anexa texto <t> ao arquivo <f>
  cd <n>            - Muda de diretorio (use '..' para voltar)
  rm <n>            - Deleta um arquivo ou diretorio vazio
  stat             - Mostra estatisticas de uso do disco
  format           - Re-formata o sistema de arquivos
SimpleFS:/$ █

```

Ao executar o comando `ls` nesse primeiro momento e no diretório base, a saída mostra apenas os diretórios padrões, mas de resto continua vazio.

```

===== SimpleFS Bare-Metal no Raspberry Pi 3 =====
Sistema de arquivos formatado e montado.
Digite 'help' para ver os comandos.
SimpleFS:/$ help
Comandos disponiveis:
  ls                - Lista arquivos e diretorios
  mkdir <n>         - Cria um novo diretorio
  touch <n>         - Cria um novo arquivo vazio
  cat <n>           - Mostra o conteudo de um arquivo
  write <f> <t>     - Escreve/anexa texto <t> ao arquivo <f>
  cd <n>            - Muda de diretorio (use '..' para voltar)
  rm <n>            - Deleta um arquivo ou diretorio vazio
  stat             - Mostra estatisticas de uso do disco
  format           - Re-formata o sistema de arquivos
SimpleFS:/$ ls
d .
d ..
d UUU@

```

Pode-se simular a criação de três novos diretórios usando o `mkdir` e ao executar o `ls` novamente, agora os novos diretórios aparecem.

```
SimpleFS:/$ mkdir dir1
SimpleFS:/$ mkdir dir2
SimpleFS:/$ mkdir dir3
SimpleFS:/$ ls
d .
d ..
d UUU@
d dir1
d dir2
d dir3
```

Agora, vamos trocar para o diretório *dir1*, executar o comando *ls* para ver que ele está vazio e vamos criar dois arquivos usando o comando *touch*, para em seguida ver eles criados dentro do diretório.

```
SimpleFS:/$ cd dir1
SimpleFS:/dir1/$ ls
d .
d ..
SimpleFS:/dir1/$ touch arq1
Arquivo 'arq1' criado.
SimpleFS:/dir1/$ touch arq2
Arquivo 'arq2' criado.
SimpleFS:/dir1/$ ls
d .
d ..
- arq1
- arq2
```

Partindo para o teste do comando *write*, pode-se realizar a escrita de uma pequena frase no arquivo *arq1* e em seguida, executar o comando *cat* para ver o conteúdo do arquivo.

```
SimpleFS:/dir1/$ write arq1 texto comprido
Texto anexado ao arquivo 'arq1'.
SimpleFS:/dir1/$ cat arq1
texto comprido
```

Agora, voltando ao diretório raiz executando *cd ..*, vamos testar o comando *rm* ao remover um diretório vazio, dentre aqueles que criamos anteriormente.

```
SimpleFS:/$ rm dir3
Item 'dir3' deletado.
SimpleFS:/$ ls
d .
d ..
d UUU@
d dir1
d dir2
- arq1
```

Nota-se que o diretório *dir3* estava vazio e foi deletado. Em seguida, executando o comando *stat* temos as estatísticas de uso da memória pelo sistema de arquivos.

```
SimpleFS:/$ stat
— Estatísticas do Sistema de Arquivos —
Inodes utilizados          6 / 128
Blocos de dados usados    73 / 8175
Espaço utilizado          37376 Bytes
Espaço total (util)       4185600 Bytes
```

Por fim, pode-se realizar a re-formatação do sistema de arquivos, por meio do comando *format*, que restaura o sistema de arquivos para o zero.

```
SimpleFS:/$ format
Formatando ...
Pronto.
SimpleFS:/$ ls
d .
d ..
d
```

Com esses testes, conclui-se, portanto, que a implementação do SimpleFileSystem ocorreu com sucesso e com resultado satisfatório.

6. Considerações Finais

O projeto atingiu com sucesso o objetivo de implementar um sistema de arquivos *bare-metal* funcional. Através do desenvolvimento prático, foi possível consolidar os conhecimentos sobre a arquitetura ARM, a interação com periféricos e os conceitos fundamentais de projeto de sistemas de arquivos, como a utilização de inodes e bitmaps. Apesar dos desafios, o resultado final é um sistema interativo e robusto que cumpre todos os requisitos propostos. Como expansão futura, poderia utilizar o cartão SD como sendo o disco do sistema de arquivos ao invés de usar uma parte da memória RAM da Raspberry Pi 3, pois dessa forma seria utilizada uma memória não-volátil em que os arquivos e diretórios teriam persistência.

- Link para o repositório: <https://github.com/vthompson27/file-system>
- Link para o vídeo no youtube:
https://www.youtube.com/watch?v=Rh0NfXfs_VE