Documentação SAR

Acadêmicos: Pedro Henrique de Moraes e Mateus Gabi Moreira

O SAR é implementado em java que tenta simular uma estrutura rudimentar de um disco rígido, onde é simulado como é armazenada cada informação, como é gerenciado os blocos livres, como podemos encontrar a informação através de indices e como tudo se comunica em um disco rigido rudimentar.
A ideia de implementação foi criar um HD, no caso a classe Disco, que é chamado pela classe Sistema que recebe através *main* SAR chamadas de sistema.
O Disco é composto por vários tipos de blocos, nós transformamos esses blocos em classes são elas: BlocoDiretório, BlocoDados, BlocoIndice, BlocoLivre, às quais todas se comunicam através da interface Bloco.
Para satisfazer a demonstração de ocorrências foi criada a classe Logger, que mostra tudo que acontece conforme o sistema executa suas ações.
A classe File auxilia duas coisas:
1 - O logger a escrever todas as ocorrências em um arquivo Log.txt
2 - Auxilia o SAR a ler o arquivo de comandos Config.txt
No trabalho foram utilizadas algumas estruturas de dados, sendo elas: Vetores, HashMaps e Listas Ligadas.
Abaixo estão descritas as classes com suas principais ideias de e seus importantes métodos em negrito.
Io.File

- Nessa classe está contida a parte do sistema que lê o arquivo Config.txt que contém as chamadas de sistemas a serem realizadas pelo usuário. O método que faz a leitura é o **readCmd(String narq)** que por sua vez chama o método encapsulado **read()**. Nesse método a string lida do Config.txt é usada no método split(), do próprio java, que separa cada palavra dividida por espaço em um vetor, onde a posição 0 do vetor é a chamada a ser realizada.
- Está contido também nessa classe o método **write()** que faz o trabalho de verificar se o arquivo Log.txt existe, se não, cria-lo e escrever os dados gerados pelo Logger no Log.txt.

Io.Logger

- O Logger serve para escrever em um arquivo e no terminal os eventos do sistema (Especificação Y)

Io.Sistema

- Contém as chamadas de sistema encapsuladas conforme Especificação N, que na verdade serão executadas pelo Models.Disco.
- Funciona como uma “interface” para o Disco, pois nele é chamado os métodos e o Sistema por sua vez manda mensagem para os métodos do Disco que são os que fazem toda a lógica do sistema.
- Método **criarArquivo (String narq, int tamarq)**, cria um arquivo de nome *narq* e tamanho *tamarq* bytes. Verifica se o arquivo já não existe e então cria uma entrada no diretório e na lista de blocos livres
- **destroiArquivo (String narq)**, destrói o arquivo de nome *narq*. Acessa o diretório para eliminar a entrada do arquivo solicitado e alterar a lista de blocos livres.
- **varreArquivo (String narq)**, varre o arquivo de nome *narq* sequencialmente desde o primeiro bloco até o último, escrevendo na tela o conteúdo de cada bloco. Acessa o diretório para realizar a varredura
- **escreveArquivo (String narq, int pos, String texto)**, escreve o texto de no máximo 100 caracteres a partir da posição *pos* do arquivo de nome *narq*. Acessa e altera o diretório para realizar a escrita a partir da posição indicada sem extrapolar *tamarq*.
- **leArquivo (String narq, int pos, int qtd)**, lê *qtd* caracteres a partir da posição *pos* do arquivo de nome *narq*. Acessa altera o diretório para realizar a leitura a partir da posição indicada sem extrapolar *tamarq*

Main.SAR

- Serve para encapsular o sistema e chama o método inicial que realiza a leitura do arquivo que contém as chamadas de sistema.

Models.Bloco

- É uma interface para comunicação entre os blocos

Models.BlocoDados

- É um bloco de dados com o vetor de caracteres onde é guardada a informação que deseja-se armazenar (dado), por exemplo um texto.

Models.BlocoDiretório

- Armazena o nome dos arquivos e seus índices e não armazena Dados em sí.

Models.BlocoIndice

- Utiliza de HashMap, estrutura Chave-Valor no qual a chave é o nome do arquivo e o valor é a lista dos índices onde ele está
- O item K da especificação é verificado nessa classe
- Numero de entradas utilizados é o tamanho do Map
- **adicionarArquivo(String narq, int indiceDesteBlocoDeIndice)**, Retorna -1 caso o bloco de índice esteja cheio ou o indice do bloco adicionado.
- **getUtilizados()**, retorna o número de índices utilizados no total.

- **isCheio()**, verifica se o bloco está cheio.
- **setIndices(LinkedList<Integer> indicesDosBlocosDeDadosReservados)**, recebe uma lista de índices e seta os que estão ocupados.
- **getIndicesDosBlocosDeDados()**, retorna os indices dos blocos de dados mais o bloco de índice. O índice do bloco de índice é o último.

Models.BlocoLivre

- Representa os blocos livres do Disco.
- Utiliza um array de indices inteiros.
- Nele setamos como -1 o primeiro Bloco (Bloco 0) pois está ocupado por ser o diretório, setamos também o Bloco 1 que é o próprio bloco que diz quais blocos estão livres. Posteriormente setamos todos os outros blocos com -1 para dizer que estão vazios. Item i da especificação do trabalho.
- **getProximoDisponivel()**, retorna o próximo índice disponível, retorna -1 caso o disco estiver cheio.
- **setIndicesComoOcupados(LinkedList<Integer> indicesDosBlocosDeDadosReservados)**, recebe uma lista ligada de índices para serem setados como ocupados.
- **getIndicesLivres()**, retorna uma lista de inteiros livres

Models.Disco

- Manipula basicamente todas as informações e faz as chamadas de cada bloco.
- Feita a leitura com a classe File, o disco é startado com os parâmetros d=numero de blocos e b=tamanho de cada bloco em bytes.
- Possui um atributo vetor de Blocos que contém todos os blocos do sistema.
- O vetor comentado no item anterior na posição 0 é transformado no bloco de diretório pois recebe um objeto do tipo BlocoDiretório nessa posição, o segundo bloco na posição 1 é o bloco com uma lista de blocos livres pois recebe um tipo BlocoLivre.
- Do bloco 2 ao d-1 são todos blocos de dados ou blocos de índices.
- **existe(String narq)**, verifica se o disco contém um arquivo com o nome passado no argumento *narq* e retorna true caso seja verdadeiro.
- **adicionarArquivo(String narq, int tamarq)**, adiciona um arquivo *narq* de tamanho *tamarq* bytes.
- No método citado anteriormente são feitas as seguintes verificações:

1. A alocação de blocos então realizada via alocação indexada usando um bloco de índice para cada arquivo. Então temos que encontrar o primeiro espaço disponível no vetor e inserir um bloco de índice. Pegamos a lista de indices disponíveis para isso.
2. Criamos um bloco de índices e então inserimos o arquivo Bloco do Diretório.

3. Reservar índices nesse *array* para o meu arquivo. Número máximo de posições vagas para um arquivo: *d - 3*. No qual cada bloco possui *b* bytes no máximo. Logo, o número de blocos que devem ser reservados para cada arquivo deve ser o teto de *tamarq / b*
4. Agora tenho que verificar se blocosParaSeremReservados é maior que a quantidade de blocos livres, se for maior não há espaço então devemos apagar o bloco de índice. Caso contrário, há espaço e devemos reservar espaço para os blocos.
5. Começamos em um for com *i* = 1 pois o 0 é o Bloco de índice, pegamos o termo *i* dos indicesLivres e adicionamos na lista ligada de índices reservados.
6. Não podemos esquecer de adicionar ao bloco de índices os índices dos blocos de dados, para isso criamos o bloco de dados.
7. Não podemos também esquecer de adicionar o índice onde está o bloco de indices deste arquivo.

- **destroiArquivo(String narq)**, método que remove um arquivo de nome *narq*

1. Se o arquivo não existe mandamos para o Log guardar que ele não existe.
2. Se o arquivo existe temos que setar como nulo: Seus blocos de dados - Seu Bloco de índice, e adicionar no bloco livre os índices: dos blocos de dados - do bloco de índice. Para isso fazemos o seguinte:
    a. Pegando o indice do bloco de indice;
    b. Inserimos o indice do bloco de indice porque ele também será nulo;
    c. Pegamos todos os indices *i* dos blocos de dados;
    d. Adicionamos esses indices na lista
    e. Tornamos nulos a posição do array
    f. Removemos o arquivo do diretorio
    g. Setamos os índíces disponíveis
3. **getQuantidadeIndicesLivres()**, retorna a quatidade de índices livres
4. **varreArquivo(String narq)**, varre o arquivo de nome narq sequencialmente desde o primeiro bloco até o último, escrevendo na tela o conteúdo de cada bloco. Acessa o diretório para realizar a varredura. Algumas observações:
    a. Para varrer um arquivo devemos: 1 - Saber se existe, 2 - Pegar os indices dos blocos de dados e 3 - Pegar os caracteres;
    b. Removemos o último bloco pois ele é o índice do próprio bloco de índice.

A importância desse trabalho consiste nos fatos de que, podemos aprender como funciona a base de um sistema de arquivos e com essa programação ainda mais especializar desafios de implementação a cada método, a cada classe e saber claro lidar com problemas como: Disco cheio, guardar mais informação que o bloco suporta, ler os arquivos de chamadas de sistema corretamente sem perda de dados e

principalmente gerar um LOG, preciso, que facilite o entendimento do usuário de como o sistema se comporta, e realiza cada ação para tornar o sistema de arquivos capaz de armazenar os dados requisitados.
A principal dificuldade junto às dificuldades descritas acima foi lidar com inúmeras restrições e como fazer um trabalho conciso que use o melhor da linguagem na qual foi desenvolvido: a orientação a objetos e a proteção de dados.
Link do repositório do trabalho: https://github.com/MateusGabi/TPSO