

Sistema de Arquivos em Java

Samara Cardoso dos Santos

Turma: 3º ADS - Manhã

Professor: Murilo Dantas

**SUMÁRIO**

1. Exercício Proposto 03
2. Classe Disk 03
3. Classe FileDescriptor 06
4. Classe FileSystem 07
5. Classe FileTable 07
6. Classe IndirectBlock 09
7. Classe Inode 09
8. Classe InodeBlock 10
9. Classe JavaFileSystem 11
10. Classe SuperBlock 17
11. Classe TestFS 17
12. Testes do Sistema de Arquivo 21
13. Referências Bibliográficas 22
14. **Exercício Proposto**

Desenvolva o projeto: Criando um sistema de arquivos. Livro-texto: p. 334-338. Este projeto consiste em projetar e implementar um sistema de arquivos simples em cima de um disco simulado. Um sistema de arquivos é um conjunto de estruturas lógicas e de rotinas, que permitem ao sistema operacional controlar o acesso ao disco rígido.A estrutura deste sistema de arquivos segue de perto baseado em UNIX sistemas de arquivos, onde um SuperBlock é usado para descrever o estado do sistema de arquivos. Internamente, cada arquivo é armazenado no sistema de arquivos usando uma estrutura de dados inode.

1. **Classe Disk**

A classe Disk.java simula o disco. O disco é um arquivo RandomAccessFile, que contém os bytes necessários para representar todos os blocos. Abaixo segue a implementação da classe com alguns comentários.

**import** java.io.\*;

**class** Disk {

// tamanho em bytes de cada bloco de disco

**public** **final** **static** **int** *BLOCK\_SIZE* = 512;

//número de blocos de disco no sistema

**public** **final** **static** **int** *NUM\_BLOCKS* = 1000;

// número de ponteiros em um bloco de disco

**public** **final** **static** **int** *POINTERS\_PER\_BLOCK* = (*BLOCK\_SIZE* / 4);

//leitura e escrita para o sitema de arquivo

**private** **int** readCount = 0;

**private** **int** writeCount = 0;

// arquivo que representa o disco simulado

**private** File fileName;

**private** RandomAccessFile disk;

// leitura do arquivo que representa o disco simulado

**public** Disk() {

**try** {

fileName = **new** File("DISK");

disk = **new** RandomAccessFile(fileName, "rw");

} **catch** (IOException e) {

System.*err*.println("Não foi possível iniciar o disco");

System.*exit*(1);

}

}

// leitura a partir do blockNum no ByteBuffer

**private** **void** seek(**int** blocknum) **throws** IOException {

**if** (blocknum < 0 || blocknum >= *NUM\_BLOCKS*)

**throw** **new** RuntimeException("Tentativa de Leitura " + blocknum

+ " está fora da faixa");

disk.seek((**long**) (blocknum \* *BLOCK\_SIZE*));

}

**public** **void** read(**int** blocknum, **byte**[] buffer) {

**if** (buffer.length != *BLOCK\_SIZE*)

**throw** **new** RuntimeException("Tamanho do buffer:" + buffer.length);

**try** {

seek(blocknum);

disk.read(buffer);

} **catch** (IOException e) {

System.*err*.println(e);

System.*exit*(1);

}

readCount++;

}

**public** **void** read(**int** blocknum, SuperBlock block) {

**try** {

seek(blocknum);

block.size = disk.readInt();

block.iSize = disk.readInt();

block.freeList = disk.readInt();

} **catch** (EOFException e) {

**if** (blocknum != 0) {

System.*err*.println(e);

System.*exit*(1);

}

block.size = block.iSize = block.freeList = 0;

} **catch** (IOException e) {

System.*err*.println(e);

System.*exit*(1);

}

readCount++;

}

**public** **void** read(**int** blocknum, InodeBlock block) {

**try** {

seek(blocknum);

**for** (**int** i = 0; i < block.node.length; i++) {

block.node[i].flags = disk.readInt();

block.node[i].owner = disk.readInt();

block.node[i].fileSize = disk.readInt();

**for** (**int** j = 0; j < 13; j++)

block.node[i].pointer[j] = disk.readInt();

}

} **catch** (IOException e) {

System.*err*.println(e);

System.*exit*(1);

}

readCount++;

}

**public** **void** read(**int** blocknum, IndirectBlock block) {

**try** {

seek(blocknum);

**for** (**int** i = 0; i < block.pointer.length; i++)

block.pointer[i] = disk.readInt();

} **catch** (IOException e) {

System.*err*.println(e);

System.*exit*(1);

}

readCount++;

}

// escrita do conteúdo do buffer no número de blocos espeficicado no disco

**public** **void** write(**int** blocknum, **byte**[] buffer) {

**if** (buffer.length != *BLOCK\_SIZE*)

**throw** **new** RuntimeException("Write: bad buffer size "

+ buffer.length);

**try** {

seek(blocknum);

disk.write(buffer);

} **catch** (IOException e) {

System.*err*.println(e);

System.*exit*(1);

}

writeCount++;

}

**public** **void** write(**int** blocknum, SuperBlock block) {

**try** {

seek(blocknum);

disk.writeInt(block.size);

disk.writeInt(block.iSize);

disk.writeInt(block.freeList);

} **catch** (IOException e) {

System.*err*.println(e);

System.*exit*(1);

}

writeCount++;

}

**public** **void** write(**int** blocknum, InodeBlock block) {

**try** {

seek(blocknum);

**for** (**int** i = 0; i < block.node.length; i++) {

disk.writeInt(block.node[i].flags);

disk.writeInt(block.node[i].fileSize);

**for** (**int** j = 0; j < 13; j++)

disk.writeInt(block.node[i].pointer[j]);

}

} **catch** (IOException e) {

System.*err*.println(e);

System.*exit*(1);

}

writeCount++;

}

**public** **void** write(**int** blocknum, IndirectBlock block) {

**try** {

seek(blocknum);

**for** (**int** i = 0; i < block.pointer.length; i++)

disk.writeInt(block.pointer[i]);

} **catch** (IOException e) {

System.*err*.println(e);

System.*exit*(1);

}

writeCount++;

}

// função que Para o disco

**public** **void** stop(**boolean** removeFile) {

System.*out*.println(generateStats());

**if** (removeFile)

fileName.delete();

}

**public** **void** stop() {

stop(**true**);

}

// Saídas de estatísticas do sistema de arquivos.

**public** String generateStats() {

**return** ("DISK: Leitura: " + readCount + " Escrita: " + writeCount);

}

1. **Classe FileDescriptor**

A classe FileDescriptor.java contém dados sobre o arquivo. Os dados que a classe tem são: inode, inumber, seek pointer, o método construtor da classe e dos respectivos getters e setters.

**class** FileDescriptor

{

**private** Inode inode;

**private** **int** inumber;

**private** **int** seekptr;

**public** FileDescriptor(Inode inode, **int** inumber){

**this**.inode = inode;

**this**.inumber = inumber;

seekptr = 0;

}

**public** Inode getInode(){

**return** inode;

}

**public** **int** getInumber(){

**return** inumber;

}

**public** **int** getSeekPointer(){

**return** seekptr;

}

**public** **void** setSeekPointer(**int** i){

seekptr = i;

}

**public** **void** setFileSize(**int** newSize){

inode.fileSize = newSize;

}

}

1. **Classe FileSystem**

A interface do sistema de arquivos a ser implementado. Tem os métodos :

**interface** FileSystem {

**public** **int** formatDisk(**int** size, **int** iSize);

**public** **int** shutdown();

**public** **int** create();

**public** **int** inumber(**int** fd);

**public** **int** open(**int** iNumber);

**public** **int** read(**int** fd, **byte**[] buffer);

**public** **int** write(**int** fd, **byte**[] buffer);

**public** **int** seek(**int** fd, **int** offset, **int** whence);

**public** **int** close(**int** fd);

**public** **int** delete(**int** iNumber);

}

1. **Classe FileTable**

A classe FileTable.java mantém o controle de todos os arquivos abertos no momento.Cada arquivo é necessário um ponteiro para a cópia da Memória de seu inode, seu número e seu ponteiro de busca atual.

**class** FileTable {

**public** **static** **final** **int** *MAX\_FILES* = 21;

// Indica se o índice especificado representa um arquivo //aberto

**public** **int** bitmap[];

**public** FileDescriptor[] fdArray;

// Contrutor da classe que instancia uma tabela do arquivo

**public** FileTable() {

fdArray = **new** FileDescriptor[*MAX\_FILES*];

bitmap = **new** **int**[*MAX\_FILES*];

**for** (**int** i = 0; i < *MAX\_FILES*; i++) {

bitmap[i] = 0;

}

}

// Retorna o índice na tabela de arquivos

**public** **int** allocate() {

**for** (**int** i = 0; i < *MAX\_FILES*; i++) {

**if** (bitmap[i] == 0)

**return** i;

}

System.*out*.println("O arquivo não pode abrir. FileTable está cheio.");

**return** -1;

}

**public** **int** add(Inode inode, **int** inumber, **int** fd) {

**if** (bitmap[fd] != 0)

**return** -1;

bitmap[fd] = 1;

fdArray[fd] = **new** FileDescriptor(inode, inumber);

**return** 0;

}

**public** **void** free(**int** fd) {

bitmap[fd] = 0;

}

// retorna true se ele é válido, false se não.

**public** **boolean** isValid(**int** fd) {

**if** (fd < 0 || fd >= *MAX\_FILES*) {

System.*out*

.println("ERRO: Arquivo Inválido (Intervalo de ser 0 <= fd <= "

+ *MAX\_FILES* + ") : " + fd);

**return** **false**;

} **else** **if** (bitmap[fd] > 0) {

**return** **true**;

} **else** {

**return** **false**;

}

}

**public** Inode getInode(**int** fd) {

**if** (bitmap[fd] == 0) {

**return** **null**;

} **else** {

**return** fdArray[fd].getInode();

}

}

**public** **int** getInumber(**int** fd) {

**if** (bitmap[fd] == 0) {

**return** 0;

} **else** {

**return** fdArray[fd].getInumber();

}

}

**public** **int** getSeekPointer(**int** fd) {

**if** (bitmap[fd] == 0) {

**return** 0;

} **else** {

**return** fdArray[fd].getSeekPointer();

}

}

**public** **int** setSeekPointer(**int** fd, **int** newPointer) {

**if** (bitmap[fd] == 0) {

**return** 0;

} **else** {

fdArray[fd].setSeekPointer(newPointer);

**return** 1;

}

}

**public** **int** setFileSize(**int** fd, **int** newFileSize) {

**if** (bitmap[fd] == 0) {

**return** 0;

} **else** {

fdArray[fd].setFileSize(newFileSize);

**return** 1;

}

}

**public** **int** getFDfromInumber(**int** inumber) {

**for** (**int** i = 0; i < *MAX\_FILES*; i++) {

**if** (bitmap[i] == 1) {

**if** (fdArray[i].getInumber() == inumber) {

**return** i;

}

}

}

**return** -1;

}

}

1. **Classe IndirectBlock**

Ao contrário dos inodes, que são fixados em número e alocados em uma parte especial do sistema de arquivos, o Indirect Block podem ser de qualquer número e são alocados na mesma parte do sistema de arquivos como blocos de dados. O número de apontadores nos blocos indiretos é dependente da dimensão e tamanho dos ponteiros do bloco do bloco.

**class** IndirectBlock {

**public** **int** pointer[] = **new** **int**[Disk.*BLOCK\_SIZE* / 4];

**public** IndirectBlock() {

clear();

}

**public** **void** clear() {

**for** (**int** i = 0; i < Disk.*BLOCK\_SIZE* / 4; i++) {

pointer[i] = 0;

}

}

**public** String toString() {

String s = **new** String();

s += "INDIRECTBLOCK:\n";

**for** (**int** i = 0; i < pointer.length; i++)

s += pointer[i] + "|";

**return** s;

}

}

1. **Classe Inode**

A classe Inode permite identificar e mapear os arquivos do disco. Possui 64 bytes cada que descreve um único arquivos. Contendo seus atributos flags, owner, fileSize e pointer.

**class** Inode {

// tamanho de um inode em bytes

**public** **final** **static** **int** *SIZE* = 64;

**int** flags;

**int** owner;

**int** fileSize;

**public** **int** pointer[] = **new** **int**[13];

**public** String toString() {

String s = "[Flags: " + flags + " Size: " + fileSize + " ";

**for** (**int** i = 0; i < 13; i++)

s += "|" + pointer[i];

**return** s + "]";

}

}

1. **Classe InodeBlock**

Cada bloco de inode contém Disk.BLOCK\_SIZE/Inode.SIZE. Os inodes são numerados consecutivamente a partir de 1. A classe InodeBlock representa esse block de inode.

**class** InodeBlock {

Inode node[] = **new** Inode[Disk.*BLOCK\_SIZE*/Inode.*SIZE*];

**public** InodeBlock() {

**for** (**int** i = 0; i < Disk.*BLOCK\_SIZE*/Inode.*SIZE*; i++)

node[i] = **new** Inode();

}

**public** String toString() {

String s = "INODEBLOCK:\n";

**for** (**int** i = 0; i < node.length; i++)

s += node[i] + "\n";

**return** s;

}

}

1. **Classe JavaFileSystem**

Essa classe implementa os dez métodos no sistema de arquivos descritos na interface FileSystem.

**import** java.util.\*;

**class** JavaFileSystem **implements** FileSystem {

**public** **static** **final** **int** *SEEK\_SET* = 0;

**public** **static** **final** **int** *SEEK\_CUR* = 1;

**public** **static** **final** **int** *SEEK\_END* = 2;

**public** **int** IBLOCK\_SIZE = 8;

**private** LinkedList<Integer> freeList;

**private** SuperBlock SuperBlock;

**private** FileTable FileTable;

**private** Disk Disk;

**private** **int** MAX\_DIRECTBLOCK\_SIZE = Disk.*BLOCK\_SIZE* \* 10;

**private** **int** SINGLE\_INDIRECTBLOCK\_SIZE = Disk.*BLOCK\_SIZE* \* Disk.*POINTERS\_PER\_BLOCK*;

**private** **int** DOUBLE\_INDIRECTBLOCK\_SIZE = SINGLE\_INDIRECTBLOCK\_SIZE \* Disk.*POINTERS\_PER\_BLOCK*;

**private** **int** TRIPLE\_INDIRECTBLOCK\_SIZE = DOUBLE\_INDIRECTBLOCK\_SIZE \* Disk.*POINTERS\_PER\_BLOCK*;

**private** **static** **int** *iNumber\_counter* = 0;

JavaFileSystem() {

Disk = **new** Disk();

FileTable = **new** FileTable();

}

**public** **int** formatDisk(**int** size, **int** iSize) {

**if** (size < iSize || size == 0 || iSize == 0) {

**return** -1;

}

SuperBlock = **new** SuperBlock();

SuperBlock.size = size;

SuperBlock.iSize = iSize;

Disk.write(0, SuperBlock);

**for** (**int** i = 1; i <= iSize; i++) {

InodeBlock InodeBlock = **new** InodeBlock();

**for** (**int** j = 0; j < InodeBlock.node.length; j++) {

InodeBlock.node[j].flags = 0;

}

Disk.write(i, InodeBlock);

}

freeList = **new** LinkedList<Integer>();

**for** (**int** i = iSize + 1; i < size; i++) {

freeList.add(i);

}

**return** 0;

}

// Fecha todos os arquivos abertos e encerra o disco simulado.

**public** **int** shutdown() {

**for** (**int** i = 0; i < FileTable.fdArray.length - 1; i++) {

**if** (FileTable.isValid(i)) {

**this**.close(i);

}

}

SuperBlock.freeList = **this**.freeList.size();

Disk.stop();

**return** 0;

}

// Cria um novo arquivo vazio.

**public** **int** create() {

**for** (**int** i = 1; i < SuperBlock.iSize; i++) {

InodeBlock InodeBlock = **new** InodeBlock();

Disk.read(i, InodeBlock);

**for** (**int** j = 1; j < InodeBlock.node.length; j++) {

Inode Inode = InodeBlock.node[j];

**if** (Inode.flags == 0) {

Inode.flags = 1;

Inode.fileSize = 0;

Inode.owner = 0;

**int** fd = FileTable.allocate();

**if** (fd != -1) {

FileTable.add(Inode, *iNumber\_counter*, fd);

*iNumber\_counter*++;

**return** fd;

}

}

}

}

**return** -1;

}

// Retorna o número de um arquivo aberto

**public** **int** inumber(**int** fd) {

**if** (!FileTable.isValid(fd))

**return** -1;

**return** FileTable.getInumber(fd) + 1;

}

// Abra um arquivo existente identificado pelo seu número

**public** **int** open(**int** iNumber) {

iNumber--;

**int** blockNumber = (iNumber / IBLOCK\_SIZE) + 1;

InodeBlock InodeBlock = **new** InodeBlock();

Disk.read(blockNumber, InodeBlock);

Inode node = InodeBlock.node[iNumber % 8];

**int** fd = FileTable.allocate();

**if** (fd != -1) {

FileTable.add(node, iNumber, fd);

}

**return** fd;

}

// Leitura do buffer aberto indicado por fd.

**public** **int** read(**int** fd, **byte**[] buffer) {

**if** (!FileTable.isValid(fd))

**return** -1;

FileDescriptor fileDescriptor = FileTable.fdArray[fd];

Inode Inode = fileDescriptor.getInode();

**if** (Inode.fileSize > Disk.*BLOCK\_SIZE* \* 10) {

**byte**[] directBuffer = **new** **byte**[Disk.*BLOCK\_SIZE* \* 10];

**int** count = readFromDirectBlock(fileDescriptor, Inode, directBuffer);

**if** (Inode.fileSize <= SINGLE\_INDIRECTBLOCK\_SIZE) {

**byte**[] indirectBuffer = **new** **byte**[Inode.fileSize

- (Disk.*BLOCK\_SIZE* \* 10)];

readFromSingleIndirectBlock(fileDescriptor, Inode, buffer);

}

} **else** {

**return** readFromDirectBlock(fileDescriptor, Inode, buffer);

}

**return** -1;

}

// Tranferencia do buffer para o arquivo a partir do atual //ponteiro de busca.

**public** **int** write(**int** fd, **byte**[] buffer) {

**if** (!FileTable.isValid(fd))

**return** -1;

FileDescriptor fileDescriptor = FileTable.fdArray[fd];

Inode Inode = fileDescriptor.getInode();

**if** (Inode.fileSize > 0) {

}

**if** (buffer.length > Disk.*BLOCK\_SIZE* \* 10) {

**byte**[] directBuffer = **new** **byte**[Disk.*BLOCK\_SIZE* \* 10];

**int** count = 0;

**for** (**int** i = 0; i < Disk.*BLOCK\_SIZE* \* 10; i++) {

directBuffer[i] = buffer[i];

count++;

}

writeToDirectBlock(fileDescriptor, Inode, directBuffer);

**if** (buffer.length <= SINGLE\_INDIRECTBLOCK\_SIZE) {

**byte**[] indirectBuffer = **new** **byte**[buffer.length];

**for** (**int** i = Disk.*BLOCK\_SIZE* \* 10; i < buffer.length; i++) {

indirectBuffer[i] = buffer[i];

count++;

}

writeToSingleIndirectBlock(fileDescriptor, Inode,

indirectBuffer);

}

**return** count;

} **else** {

**return** writeToDirectBlock(fileDescriptor, Inode, buffer);

}

}

// Modifica o ponteiro de busca.

**public** **int** seek(**int** fd, **int** offset, **int** whence) {

**int** seekPointer = FileTable.getSeekPointer(fd);

**switch** (whence) {

**case** *SEEK\_SET*:

seekPointer = offset;

**break**;

**case** *SEEK\_CUR*:

seekPointer += offset;

**break**;

**case** *SEEK\_END*:

seekPointer = FileTable.getInode(fd).fileSize + offset;

**break**;

**default**:

seekPointer = -1;

}

**if** (seekPointer < 0)

**return** -1;

**return** seekPointer;

}

**public** **int** close(**int** fd) {

**if** (!FileTable.isValid(fd))

**return** -1;

FileDescriptor fileDescriptor = FileTable.fdArray[fd];

InodeBlock InodeBlock = **new** InodeBlock();

**int** iNumber = FileTable.getInumber(fd);

**int** blockNumber = (iNumber / IBLOCK\_SIZE) + 1;

Disk.read(blockNumber, InodeBlock);

InodeBlock.node[iNumber % 8] = fileDescriptor.getInode();

Disk.write(blockNumber, InodeBlock);

FileTable.free(fd);

**return** 0;

}

// Exclua o arquivo com o número dado

**public** **int** delete(**int** iNumber) {

**if** (FileTable.isValid(FileTable.getFDfromInumber(iNumber)))

**return** -1;

**int** fd = FileTable.getFDfromInumber(iNumber);

FileDescriptor fileDescriptor = FileTable.fdArray[fd];

Inode Inode = fileDescriptor.getInode();

**for** (**int** i = 0; i <= Inode.fileSize / Disk.*BLOCK\_SIZE*; i++) {

}

**throw** **new** RuntimeException("");

}

**public** String toString() {

**throw** **new** RuntimeException("");

}

**public** **int** writeToDirectBlock(FileDescriptor fileDescriptor, Inode Inode,

**byte**[] buffer) {

**int** pointerLoc = 0;

**int** i = 0;

Inode.pointer[pointerLoc] = freeList.removeFirst();

Inode.fileSize = buffer.length;

fileDescriptor.setSeekPointer(buffer.length);

**if** (buffer.length <= 512) {

**byte**[] buf = **new** **byte**[Disk.*BLOCK\_SIZE*];

**for** (i = 0; i < buffer.length; i++) {

buf[i] = buffer[i];

}

Disk.write(Inode.pointer[pointerLoc], buf);

**return** i;

} **else** {

**int** writes = 0;

**int** count = 0;

**for** (**int** j = 0; j < buffer.length; j += 512) {

**byte**[] buf = **new** **byte**[Disk.*BLOCK\_SIZE*];

**if** (buffer.length - j >= Disk.*BLOCK\_SIZE*) {

**for** (i = 0; i < Disk.*BLOCK\_SIZE*; i++) {

buf[i] = buffer[i + j];

writes++;

}

} **else** {

**for** (i = 0; i < buffer.length - j; i++) {

buf[i] = buffer[i + j];

writes++;

}

}

Disk.write(Inode.pointer[count], buf);

count++;

}

**return** writes;

}

}

**public** **int** writeToSingleIndirectBlock(FileDescriptor fileDescriptor,

Inode Inode, **byte**[] buffer) {

**int** pointerLoc = 10;

**int** i = 0;

IndirectBlock IndirectBlock = **new** IndirectBlock();

Inode.pointer[pointerLoc] = IndirectBlock.pointer[0];

**int** count = 0;

**int** writes = 0;

**for** (**int** j = 0; j <= buffer.length; j += 512) {

**byte**[] buf = **new** **byte**[Disk.*BLOCK\_SIZE*];

**if** (buffer.length - j >= Disk.*BLOCK\_SIZE*) {

**for** (i = 0; i < Disk.*BLOCK\_SIZE*; i++) {

buf[i] = buffer[i + j];

writes++;

}

} **else** {

**for** (i = 0; i < buffer.length - j; i++) {

buf[i] = buffer[i + j];

writes++;

}

}

Disk.write(IndirectBlock.pointer[count], buf);

count++;

}

Inode.fileSize = buffer.length;

fileDescriptor.setSeekPointer(buffer.length);

Disk.write(Inode.pointer[pointerLoc], IndirectBlock);

**return** writes;

}

**public** **int** writeToDoubleIndirectBlock(FileDescriptor fileDescriptor,

Inode Inode, **byte**[] buffer) {

**int** pointerLoc = 11;

**return** 0;

}

**public** **int** writeToTripleIndirectBlock(FileDescriptor fileDescriptor,

Inode Inode, **byte**[] buffer) {

**int** pointerLoc = 12;

**return** 0;

}

**public** **int** readFromDirectBlock(FileDescriptor fileDescriptor, Inode Inode,

**byte**[] buffer) {

**int** pointerLoc = 0;

**byte**[] buf = **new** **byte**[Disk.*BLOCK\_SIZE*];

Disk.read(Inode.pointer[pointerLoc], buf);

**for** (**int** i = fileDescriptor.getSeekPointer(); i < buffer.length; i++) {

buffer[i] = buf[i];

}

**return** Inode.fileSize;

}

**public** **int** readFromSingleIndirectBlock(FileDescriptor fileDescriptor,

Inode Inode, **byte**[] buffer) {

**int** pointerLoc = 10;

IndirectBlock IndirectBlock = **new** IndirectBlock();

Disk.read(Inode.pointer[pointerLoc], IndirectBlock);

**int** count = 0;

**int** reads = 0;

**for** (**int** i = fileDescriptor.getSeekPointer(); i < Inode.fileSize; i += 512) {

**byte**[] buf = **new** **byte**[Disk.*BLOCK\_SIZE*];

**for** (**int** j = i; j < Disk.*BLOCK\_SIZE*; j++) {

buf[i] = buffer[i + j];

reads++;

}

Disk.read(IndirectBlock.pointer[count], buf);

count++;

}

**return** reads;

}

**public** **int** readFromDoubleIndirectBlock(FileDescriptor fileDescriptor,

Inode Inode, **byte**[] buffer) {

**int** pointerLoc = 11;

**return** -1;

}

**public** **int** readFromTripleIndirectBlock(FileDescriptor fileDescriptor,

Inode Inode, **byte**[] buffer) {

**int** pointerLoc = 12;

**return** -1;

}

}

1. **Classe SuperBlock**

A classe SuperBlock possui as informações sobre a estrutura do sistema de arquivos, contendo o número de blocos no sistema de arquivo, o nu mero de blocos de índice e o primeiro bloco na lista de

**class** SuperBlock {

**int** size;

**int** iSize;

**int** freeList;

**public** String toString () {

**return**

"SUPERBLOCK:\n"

+ "Size: " + size

+ " Isize: " + iSize

+ " freeList: " + freeList;

}

}

1. **Classe TestFS**

A classe TestFS contém o método principal que permitirá o teste da interface FileSystem.java com os métodos para a manipulação do sistema de arquivo.

**import** java.io.\*;

**import** java.util.\*;

**class** TestFS {

**private** **static** FileSystem *fs* = **new** JavaFileSystem();

**private** **static** Hashtable *vars* = **new** Hashtable();

**public** **static** **void** main(String[] args) {

**if** (args.length > 1)

System.*err*.println("Uso: Testes [arquivo]");

**boolean** fromFile = (args.length == 1);

BufferedReader data = **null**;

**if** (fromFile) {

**try** {

data = **new** BufferedReader(**new** FileReader(**new** File(args[0])));

} **catch** (FileNotFoundException e) {

System.*err*.println("Erro: Arquivo " + args[0]

+ " não encontrado.");

System.*exit*(1);

}

} **else**

data = **new** BufferedReader(**new** InputStreamReader(System.*in*));

**for** (;;) {

**try** {

**if** (!fromFile) {

System.*out*.print("--> ");

System.*out*.flush();

}

String line = data.readLine();

**if** (line == **null**)

System.*exit*(0);

line = line.trim();

**if** (line.length() == 0) {

System.*out*.println();

**continue**;

}

**if** (line.startsWith("//")) {

**if** (fromFile)

System.*out*.println(line);

**continue**;

}

**if** (line.startsWith("/\*"))

**continue**;

**if** (fromFile)

System.*out*.println("--> " + line);

String target = **null**;

**int** equals = line.indexOf('=');

**if** (equals > 0) {

target = line.substring(0, equals).trim();

line = line.substring(equals + 1).trim();

}

StringTokenizer cmds = **new** StringTokenizer(line);

String cmd = cmds.nextToken();

**int** result = 0;

**if** (cmd.equalsIgnoreCase("formatDisk")

|| cmd.equalsIgnoreCase("format")) {

**int** arg1 = *nextValue*(cmds);

**int** arg2 = *nextValue*(cmds);

result = *fs*.formatDisk(arg1, arg2);

} **else** **if** (cmd.equalsIgnoreCase("shutdown")) {

result = *fs*.shutdown();

} **else** **if** (cmd.equalsIgnoreCase("create")) {

result = *fs*.create();

} **else** **if** (cmd.equalsIgnoreCase("open")) {

result = *fs*.open(*nextValue*(cmds));

} **else** **if** (cmd.equalsIgnoreCase("inumber")) {

result = *fs*.inumber(*nextValue*(cmds));

} **else** **if** (cmd.equalsIgnoreCase("read")) {

**int** arg1 = *nextValue*(cmds);

**int** arg2 = *nextValue*(cmds);

result = *readTest*(arg1, arg2);

} **else** **if** (cmd.equalsIgnoreCase("write")) {

**int** arg1 = *nextValue*(cmds);

String arg2 = cmds.nextToken();

**int** arg3 = *nextValue*(cmds);

result = *writeTest*(arg1, arg2, arg3);

} **else** **if** (cmd.equalsIgnoreCase("seek")) {

**int** arg1 = *nextValue*(cmds);

**int** arg2 = *nextValue*(cmds);

**int** arg3 = *nextValue*(cmds);

result = *fs*.seek(arg1, arg2, arg3);

} **else** **if** (cmd.equalsIgnoreCase("close")) {

result = *fs*.close(*nextValue*(cmds));

} **else** **if** (cmd.equalsIgnoreCase("delete")) {

result = *fs*.delete(*nextValue*(cmds));

} **else** **if** (cmd.equalsIgnoreCase("quit")) {

System.*exit*(0);

} **else** **if** (cmd.equalsIgnoreCase("vars")) {

**for** (Enumeration e = *vars*.keys(); e.hasMoreElements();) {

Object key = e.nextElement();

Object val = *vars*.get(key);

System.*out*.println("\t" + key + " = " + val);

}

**continue**;

} **else** {

System.*out*.println("Comando inválido");

**continue**;

}

**if** (target == **null**)

System.*out*.println(" Resultado: " + result);

**else** {

*vars*.put(target, **new** Integer(result));

System.*out*.println(" " + target + " = " + result);

}

}

**catch** (NumberFormatException e) {

System.*out*.println("Argumento Incorreto");

} **catch** (NoSuchElementException e) {

System.*out*.println("Número incorreto de elementos");

} **catch** (IOException e) {

System.*err*.println(e);

}

}

}

**static** **private** **int** nextValue(StringTokenizer cmds) {

String arg = cmds.nextToken();

Object val = *vars*.get(arg);

**return** (val == **null**) ? Integer.*parseInt*(arg) : ((Integer) val)

.intValue();

}

**private** **static** **int** readTest(**int** fd, **int** size) {

**byte**[] buffer = **new** **byte**[size];

**int** length;

**for** (**int** i = 0; i < size; i++)

buffer[i] = (**byte**) '\*';

length = *fs*.read(fd, buffer);

**for** (**int** i = 0; i < length; i++)

*showchar*(buffer[i]);

**if** (length != -1)

System.*out*.println();

**return** length;

}

**private** **static** **int** writeTest(**int** fd, String str, **int** size) {

**byte**[] buffer = **new** **byte**[size];

**for** (**int** i = 0; i < buffer.length; i++)

buffer[i] = (**byte**) str.charAt(i % str.length());

**return** *fs*.write(fd, buffer);

}

**private** **static** **void** showchar(**byte** b) {

**if** (b < 0) {

System.*out*.print("M-");

b += 0x80;

}

**if** (b >= ' ' && b <= '~') {

System.*out*.print((**char**) b);

**return**;

}

**switch** (b) {

**case** '\0':

System.*out*.print("\\0");

**return**;

**case** '\n':

System.*out*.print("\\n");

**return**;

**case** '\r':

System.*out*.print("\\r");

**return**;

**case** '\b':

System.*out*.print("\\b");

**return**;

**case** 0x7f:

System.*out*.print("\\?");

**return**;

**default**:

System.*out*.print("^" + (**char**) (b + '@'));

**return**;

}

}

}

1. **Testes do Sistema de Arquivo**

Teste do Sistema de Arquivo testando os dez métodos exigidos na implementação de File System. Também testa a verificação de erro no código.

**Teste com métodos da interface FileSystem.java**

--> formatDisk 100 10

--> file1 = create

--> inum1 = inumber file1

--> write file1 HiThere 10000

--> seek file1 -10 1

--> read file1 50

--> file2 = create

--> inum2 = inumber file2

--> write file2 Names\_ 512

--> file3 = create

--> inum3 = inumber file3

--> file2 = open inum1

--> close file1

--> delete inum2

--> shutdown

--> quit

**Teste com Verificação de Erro**

--> formatDisk 100 10

--> file1 = create

--> open 21

--> close 21

--> close -1

--> write file1 Projeto de SO 2048

--> write file1 SO 2048

--> write file4 Aaaaaayyyyy!\_ 512

--> read file2 100

--> close file2

--> inum1 = inumber file2

--> shutdown

--> quit

1. **Referências Bibliográficas**

Carlos Eduardo Maiolino, Sistemas de arquivos no Linux. Disponível em: <http://people.redhat.com/~cmaiolin/talks/filesystem\_unicamp.pdf>.Acesso em: 21 de junho de 2014.

Guilherme Silveira, Aprendendo a manipular arquivos. Disponível em: <http://www.guj.com.br/articles/13>.Acesso em 19 de junho de 2014

Oracle, Command-Line Arguments.Disponível em: <<http://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html>>. Acesso em: 19 de junho de 2014.

Oracle, File. Disponível em: <http://docs.oracle.com/javase/7/docs/api/java/io/File.html>. Acesso em 19 de junho de 2014.

Oracle, FileSystems. Disponível em: <http://docs.oracle.com/javase/7/docs/api/java/nio/file/FileSystems.html>. Acesso em: 20 de junho de 2014.

SILBERSCHATZ, Abraham, Sistemas Operacionais com JAVA. 7ª edição. Campus: São Paulo, 2008.

Taisy Silva Weber, Sistema de Arquivos UNIX. Disponível em:< http://www.cesarkallas.net/arquivos/apostilas/programacao/c\_c%2B%2B/c/PL06LinuxFS.PDF>. Acesso em: 21 de junho de 2014.