Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Системне програмне забезпечення – 2

Лабораторна робота №2

**«Файлова система (частина 1)»**

Виконала:

студентка групи ІВ-71

Молчанова В.С.

Перевірив:

ст. вик. Сімоненко А.В.

Київ

2020 р.

### Опис ідеї

Є об'єкт файлової системи, який має блоки пам'яті, масив дескрипторів файлів. Дескриптори мають посилання на блоки пам'яті, які використовуються для зберігання файлів

### Структури даних

*Class DataBlock* – зберігає данні або посилання на файли

*Class Descriptor* – зберігає дані про файл (розмір, флаг isFolder, кількість посилань на файл) та посилання на блоки з даними файлу

*Class File System* – містить кореневу директорію, масив блоків даних, карту використання блоків, масив дескрипторів файлів та масив відкритих дескрипторів.

### Лістинг програми

DataBlock.kt

import java.util.HashMap

import java.io.Serializable

class DataBlock : Serializable {

var data: String = " ".repeat(BLOCK\_SIZE)

val dataList: HashMap<String, DescriptorIndex> = HashMap()

fun setData(name: String, id: Int) {

if (dataList.size < BLOCK\_SIZE) {

dataList[name] = id

} else {

println("Failed to add link at the block")

}

}

}

Descriptor.kt

import java.util.ArrayList

import java.io.Serializable

class Descriptor(

var isFolder: Boolean,

var refCount: Int) : Serializable {

constructor(): this(false, 0)

val size: Int

get() {return linksToBlocks.count()}

val linksToBlocks = ArrayList<BlockIndex>()

fun addLinkToBlock(blockIndex: BlockIndex) {

if (linksToBlocks.count() < MAX\_LINK\_COUNT)

linksToBlocks.add(blockIndex)

else

println("Cannot add new link to this file")

}

}

FileSystem.kt

import java.util.HashMap

import java.util.ArrayList

import java.io.Serializable

const val MAX\_BLOCK\_COUNT = 8

const val MAX\_FILE\_COUNT = 4

const val BLOCK\_SIZE = 8

const val MAX\_LINK\_COUNT = 5

typealias DescriptorIndex = Int

typealias BlockIndex = Int

class FileSystem : Serializable {

val folder: Descriptor = Descriptor(true, 1)

val dataBlocks = ArrayList(List(MAX\_BLOCK\_COUNT){DataBlock()})

val usedBlocks = ArrayList(List(MAX\_BLOCK\_COUNT){false})

val descriptors = ArrayList<Descriptor>(List(MAX\_FILE\_COUNT){Descriptor()})

val openedDescriptors = HashMap<Int, DescriptorIndex>()

fun fileStat(fdIdx: DescriptorIndex) {

if (fdIdx in 0..(MAX\_FILE\_COUNT - 1)) {

val fd = descriptors[fdIdx]

println("RefCount = ${fd.refCount}, size = ${fd.size}, links to file blocks: ${fd.linksToBlocks}")

} else {

println("There is no descriptor with id $fdIdx")

}

}

fun createFile(fileName: String) {

val fdIdx = getEmptyDescriptor()

if (fdIdx == null) {

println("There are no empty descriptors, file creation failed")

return

}

val fd = Descriptor(false, 1)

descriptors[fdIdx] = fd

val block = getFirstFreeBlock() ?: return

folder.addLinkToBlock(block)

usedBlocks[block] = true

dataBlocks[block].setData(fileName, fdIdx)

println("New file created: $fileName")

}

fun openFile(fileName: String){

val fd = getFileDescriptor(fileName) ?: return

val fdId = getFdId()

openedDescriptors[fdId] = fd

println("File [$fileName] was opened FD = $fdId")

}

fun closeFile(fdId: Int){

openedDescriptors.remove(fdId)

println("Completed")

}

fun ls() {

println("Files:")

for (i in 0 until folder.linksToBlocks.size) {

val adrOfBlock = folder.linksToBlocks[i]

val h = dataBlocks[adrOfBlock].dataList

println(h)

}

}

fun readFile(fdId: Int, offset: Int, size: Int) {

val fdIdx = openedDescriptors[fdId]

if (fdIdx == null) {

println("There isn't opened file with FD = $fdId")

return

}

val fd = descriptors[fdIdx]

var data = ""

for (blockIdx in fd.linksToBlocks) {

data += dataBlocks[blockIdx].data

}

data = data.substring(offset, offset + size)

println("Result: \n $data")

}

fun writeToFile(fdId: Int, offset: Int, size: Int, inputData: String){

var data = inputData

if (data.length > size)

data = data.substring(0, size)

val fdIdx = openedDescriptors[fdId]

if (fdIdx == null) {

println("There isn't opened file with FD = $fdId")

return

}

val fd = descriptors[fdIdx]

val startBlock = offset / BLOCK\_SIZE

var endBlock = (offset + size) / BLOCK\_SIZE

if (endBlock > MAX\_BLOCK\_COUNT)

endBlock = MAX\_BLOCK\_COUNT - 1

while (fd.linksToBlocks.count() < endBlock + 1) {

val blockIdx = getFirstFreeBlock() ?: return

fd.addLinkToBlock(blockIdx)

usedBlocks[blockIdx] = true

}

val startIdx = offset % BLOCK\_SIZE

val endIdx = (offset + size) % BLOCK\_SIZE

if (startBlock == endBlock) {

val block = dataBlocks[fd.linksToBlocks[startBlock]]

block.data = block.data.substring(0 until startIdx) +

data + block.data.substring(endIdx until BLOCK\_SIZE)

} else {

for (i in startBlock..endBlock) {

val block = dataBlocks[fd.linksToBlocks[i]]

if (i == startBlock) {

block.data = block.data.substring(0 until startIdx) +

data.substring(0 until BLOCK\_SIZE - startIdx)

data = data.drop(BLOCK\_SIZE - startIdx)

} else if (i == endBlock) {

block.data = data +

block.data.substring(endIdx until BLOCK\_SIZE)

} else {

block.data = data.take(BLOCK\_SIZE)

data = data.drop(BLOCK\_SIZE)

}

}

}

println("Completed")

}

fun link(fileName: String, linkName: String){

val block = getFirstFreeBlock() ?: return

folder.linksToBlocks.add(block)

val fd = getFileDescriptor(fileName) ?: return

usedBlocks[block] = true

dataBlocks[block].setData(linkName, fd)

descriptors[fd].refCount += 1

println("$fileName linked to $linkName")

}

fun unlink(linkName: String) {

val fdIdx = getFileDescriptor(linkName) ?: return

val fd = descriptors[fdIdx]

fd.refCount -= 1

for (i in 0 until folder.linksToBlocks.size) {

if (dataBlocks[folder.linksToBlocks[i]].dataList.containsKey(linkName)) {

dataBlocks[folder.linksToBlocks[i]].dataList.remove(linkName)

folder.linksToBlocks.removeAt(i)

break

}

}

if ( fd.refCount == 0) {

for (i in 0 until fd.size) {

val blockIdx = fd.linksToBlocks[i]

dataBlocks[blockIdx].data = " ".repeat(BLOCK\_SIZE)

usedBlocks[blockIdx] = false

}

descriptors[fdIdx] = Descriptor()

}

println("Completed")

}

fun truncate(fileName: String, size: Int){

val fdIdx = getFileDescriptor(fileName) ?: return

val fd = descriptors[fdIdx]

val endBlockInFd = size / BLOCK\_SIZE

val fileBlocksCount = fd.size

if (endBlockInFd + 1 > fileBlocksCount) {

for (i in fileBlocksCount..endBlockInFd) {

val blockIdx = getFirstFreeBlock() ?: return

usedBlocks[blockIdx] = true

val block = dataBlocks[blockIdx]

block.data = " ".repeat(BLOCK\_SIZE)

}

} else {

val endBlock = dataBlocks[fd.linksToBlocks[endBlockInFd]]

val bytesUnEndBlock = size % BLOCK\_SIZE

endBlock.data = endBlock.data.take(bytesUnEndBlock) + " ".repeat(BLOCK\_SIZE - bytesUnEndBlock)

for (i in endBlockInFd + 1 until fileBlocksCount) {

val blockIdx = fd.linksToBlocks[i]

usedBlocks[blockIdx] = false

val block = dataBlocks[blockIdx]

block.data = " ".repeat(BLOCK\_SIZE)

}

}

println("Completed")

}

private fun getEmptyDescriptor(): DescriptorIndex? {

val fdIdx = descriptors.indexOfFirst { it.refCount == 0 }

return if (fdIdx == -1) null else fdIdx

}

private fun getFirstFreeBlock(): BlockIndex? {

val blockIdx = usedBlocks.indexOfFirst { !it }

return if (blockIdx == -1) {

println("There are no free linksToBlocks, operation failed.")

null

} else {

blockIdx

}

}

private fun getFileDescriptor(fileName: String): DescriptorIndex? {

var fd: DescriptorIndex? = null

for (i in 0 until folder.linksToBlocks.size) {

val adrOfBlock = folder.linksToBlocks[i]

val h = dataBlocks[adrOfBlock].dataList

if (h.containsKey(fileName)) {

fd = h[fileName]

break

}

}

if (fd == null) {

println("File $fileName doesn't exist in the current file system.")

}

return fd

}

private fun getFdId(): Int {return if (openedDescriptors.isEmpty()) 0 else (openedDescriptors.keys.max()!! + 1)}

}

Main.kt

import java.io.ObjectOutputStream

import java.io.FileOutputStream

import java.io.File

import java.io.ObjectInputStream

import java.io.FileInputStream

import java.io.IOException

import java.util.\*

const val WRONG\_ARG\_NUMBER\_EXCEPTION = "Wrong number of arguments."

const val EXT = ".fs"

object Main {

var scan = Scanner(System.`in`)

lateinit var fileSystem: FileSystem

fun mount(args: List<String>) {

when (args.count()) {

0 -> {

fileSystem = FileSystem()

println("Mounted empty file system.")

}

1 -> {

val fileName = args[0]

if (!fileName.contains(EXT)) {

println("Wrong type of file, mount failed.")

return

}

val file = File(fileName)

if (file.exists()) {

val fileIn: FileInputStream

try {

fileIn = FileInputStream(file.absolutePath)

val in1 = ObjectInputStream(fileIn)

fileSystem = in1.readObject() as FileSystem

in1.close()

fileIn.close()

} catch (e: Exception) {

println("Mount failed.")

e.printStackTrace()

}

println("Mounted file system from file $fileName.")

} else {

println("File $fileName doesn't exist")

}

}

else -> { println(WRONG\_ARG\_NUMBER\_EXCEPTION) }

}

}

fun unmount(args: List<String>) {

if (checkArgsCount(args, 1)) {

val fileName = args[0]

val myFile = File(fileName)

val fileOut: FileOutputStream

try {

fileOut = FileOutputStream(myFile.path + if (fileName.contains(EXT)) "" else EXT)

val out = ObjectOutputStream(fileOut)

out.writeObject(fileSystem)

out.close()

fileOut.close()

} catch (e: Exception) {

println("Unmount failed.")

e.printStackTrace()

}

}

}

fun filestat(args: List<String>) {

if (checkArgsCount(args, 1)) {

val id = Integer.parseInt(args[0])

fileSystem.fileStat(id)

}

}

fun create(args: List<String>) {

if (checkArgsCount(args, 1)) {

val fileName = args[0]

fileSystem.createFile(fileName)

}

}

fun ls(args: List<String>) {

if (checkArgsCount(args, 0)){

fileSystem.ls()

}

}

fun open(args: List<String>) {

if (checkArgsCount(args, 1)) {

val fileName = args[0]

fileSystem.openFile(fileName)

}

}

fun close(args: List<String>) {

if (checkArgsCount(args, 1)) {

val fdIdx = Integer.parseInt(args[0])

fileSystem.closeFile(fdIdx)

}

}

fun read(args: List<String>) {

if (checkArgsCount(args, 3)) {

val fdIdx = Integer.parseInt(args[0])

val offset = Integer.parseInt(args[1])

val size = Integer.parseInt(args[2])

fileSystem.readFile(fdIdx, offset, size)

}

}

fun write(args: List<String>) {

if (checkArgsCount(args, 4)) {

val fdIdx = Integer.parseInt(args[0])

val offset = Integer.parseInt(args[1])

val size = Integer.parseInt(args[2])

val inputData = args[3]

fileSystem.writeToFile(fdIdx, offset, size, inputData)

}

}

fun link(args: List<String>) {

if (checkArgsCount(args, 2)) {

val fileName = args[0]

val linkName = args[1]

fileSystem.link(fileName, linkName)

}

}

fun unlink(args: List<String>) {

if (checkArgsCount(args, 1)) {

val linkName = args[0]

fileSystem.unlink(linkName)

}

}

fun truncate(args: List<String>) {

if (checkArgsCount(args, 2)) {

val fileName = args[0]

val size = Integer.parseInt(args[1])

fileSystem.truncate(fileName, size)

}

}

@JvmStatic

fun main(globalArgs: Array<String>) {

while (true) {

val input = scan.nextLine()

val (command, args) = getCommandAndArgs(input)

try {

when (command) {

"mount" -> mount(args)

"unmount" -> unmount(args)

"filestat" -> filestat(args)

"create" -> create(args)

"ls" -> ls(args)

"open" -> open(args)

"close" -> close(args)

"read" -> read(args)

"write" -> write(args)

"link" -> link(args)

"unlink" -> unlink(args)

"truncate" -> truncate(args)

else -> {println("Unknown command")}

}

} catch (e: IOException) {

e.printStackTrace()

}

}

}

private fun checkArgsCount(args: List<String>, n: Int): Boolean {

if (args.count() != n) {

println(WRONG\_ARG\_NUMBER\_EXCEPTION)

return false

}

return true

}

private fun getCommandAndArgs(input: String): Pair<String, List<String>> {

val args = input.split(" ").toMutableList()

val command = args.removeAt(0)

return command to args

}