Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Системне програмне забезпечення – 2

Лабораторна робота №3

**«Файлова система (частина 2)»**

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. Дескриптор директорій зберігає посилання на файли цієї директорії у двох блоках, у першому знаходиться назва файлу, другому – індекс його дескриптора. Символічні посилання зберігаються у файлі, назва якого це ім’я посилання, а блоки даних зберігають у собі кроки шляху, на яке воно посилається.

### 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.io.Serializable

class DataBlock : Serializable {

var data = ByteArray(BLOCK\_SIZE)

init {

fillWithBlanks(0 until BLOCK\_SIZE)

}

fun writeData(input: String, offset: Int = 0) {

for ((idx, i) in (offset until BLOCK\_SIZE).withIndex()) {

data[i] = input[idx].toByte()

if (idx == input.length - 1)

break

}

}

fun fillWithBlanks(range: IntRange) {

for (i in range) {

data[i] = EMPTY\_CHAR.toByte()

}

}

}

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 = 200

const val MAX\_DESCRIPTOR\_COUNT = 10

const val BLOCK\_SIZE = 8

const val MAX\_LINK\_COUNT = 10

const val EMPTY\_CHAR = '`'

const val sep = '/'

val charset = Charsets.UTF\_8

typealias DescriptorIndex = Int

typealias BlockIndex = Int

class FileSystem : Serializable {

private val root: DescriptorIndex = 0

var path = mutableListOf("root")

private var currentDirectory: DescriptorIndex = root

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

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

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

private val openedDescriptors = HashMap<Int, DescriptorIndex>()

init {

val rootFd = Descriptor(true, 1)

descriptors[root] = rootFd

addLinkAtDirectory(rootFd, "..", root)

addLinkAtDirectory(rootFd, ".", root)

}

fun fileStat(fdIdx: DescriptorIndex) {

if (fdIdx in 0..(MAX\_DESCRIPTOR\_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): DescriptorIndex? {

val fdIdx = getEmptyDescriptor()

if (fdIdx == null) {

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

return null

}

val fd = Descriptor(false, 1)

descriptors[fdIdx] = fd

val parentDir = descrPathLookup(fileName) ?: return null

addLinkAtDirectory(descriptors[parentDir], getFileNameFromPath(fileName), fdIdx)

return fdIdx

}

fun openFile(fileName: String){

val fd = descrLookup(fileName) ?: return

val fdId = getFdId()

openedDescriptors[fdId] = fd

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

}

fun closeFile(fdId: Int){

openedDescriptors.remove(fdId)

println("Completed")

}

fun ls() {

val currentDir = descriptors[currentDirectory]

val files = HashMap<String, Int>()

val dirs = HashMap<String, Int>()

for (i in 0 until currentDir.linksToBlocks.size step 2) {

val nameBlock = dataBlocks[currentDir.linksToBlocks[i]]

val linkBlock = dataBlocks[currentDir.linksToBlocks[i+1]]

val name = getFileName(nameBlock)

val fdIdx = getFdIdx(linkBlock)

val fd = descriptors[fdIdx]

if (fd.isFolder) {

dirs[name] = fdIdx

} else {

files[name] = fdIdx

}

}

println("Directories:")

for (dir in dirs) {

println("\t$dir")

}

println("Files:")

for (file in files) {

println("\t$file")

}

}

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.toString(charset)

}

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)

}

val startIdx = offset % BLOCK\_SIZE

if (startBlock == endBlock) {

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

block.writeData(data, startIdx)

} else {

for (i in startBlock..endBlock) {

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

when (i) {

startBlock -> {

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

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

}

endBlock -> block.writeData(data, 0)

else -> {

block.writeData(data)

data = data.drop(BLOCK\_SIZE)

}

}

}

}

println("Completed")

}

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

val fd = descrLookup(fileName) ?: return

descriptors[fd].refCount += 1

val parentDir = descrPathLookup(fileName) ?: return

addLinkAtDirectory(descriptors[parentDir], linkName, fd)

println("${getFileNameFromPath(fileName)} linked to $linkName")

}

fun unlink(linkName: String) {

val fdIdx = descrLookup(linkName) ?: return

val parentDirIdx = descrPathLookup(linkName) ?: return

val parentDir = descriptors[parentDirIdx]

val fd = descriptors[fdIdx]

fd.refCount -= 1

for (i in 0 until parentDir.linksToBlocks.size step 2) {

val nameBlock = dataBlocks[parentDir.linksToBlocks[i]]

if (getFileName(nameBlock) == linkName) {

parentDir.linksToBlocks.removeAt(i)

parentDir.linksToBlocks.removeAt(i)

break

}

}

if ( fd.refCount == 0) {

for (i in 0 until fd.size) {

val blockIdx = fd.linksToBlocks[i]

dataBlocks[blockIdx].fillWithBlanks(0 until BLOCK\_SIZE)

usedBlocks[blockIdx] = false

}

descriptors[fdIdx] = Descriptor()

}

println("Completed") }

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

val fdIdx = descrLookup(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

val block = dataBlocks[blockIdx]

block.fillWithBlanks(0 until BLOCK\_SIZE)

}

} else {

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

val bytesUnEndBlock = size % BLOCK\_SIZE

endBlock.fillWithBlanks(bytesUnEndBlock until 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.fillWithBlanks(0 until BLOCK\_SIZE)

}

}

println("Completed")

}

fun mkdir(dirName: String) {

val fdIdx = getEmptyDescriptor()

if (fdIdx == null) {

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

return

}

val fd = Descriptor(true, 1)

descriptors[fdIdx] = fd

val parentDir = descrPathLookup(dirName) ?: return

addLinkAtDirectory(descriptors[parentDir], getFileNameFromPath(dirName), fdIdx)

createDirectoryLinks(parentDir, fdIdx)

println("Directory created.")

}

fun rmdir(dirName: String) {

val fdIdx = descrLookup(dirName) ?: return

val parentDirIdx = descrPathLookup(dirName) ?: return

val fd = descriptors[fdIdx]

val parentDir = descriptors[parentDirIdx]

for (i in 0 until fd.linksToBlocks.size step 2) {

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

val name = getFileName(nameBlock)

if (name != "." && name != "..") {

println("Directory is not empty, deletion failed.")

return

}

}

val nameWithoutPath = getFileNameFromPath(dirName)

for (i in 0 until parentDir.linksToBlocks.size step 2) {

val nameBlock = dataBlocks[parentDir.linksToBlocks[i]]

if (getFileName(nameBlock) == nameWithoutPath) {

parentDir.linksToBlocks.removeAt(i)

parentDir.linksToBlocks.removeAt(i)

break

}

}

descriptors[fdIdx] = Descriptor()

println("Completed")

}

fun cd(dirName: String) {

val fdIdx = descrLookup(dirName, currentDirectory, true) ?: return

currentDirectory = fdIdx

}

fun cd() {

currentDirectory = root

path = mutableListOf("root")

}

fun symlink(pathName: String, linkName: String) {

val fdIdx = createFile(linkName) ?: return

val steps = pathName.split(sep)

if (steps.count() > MAX\_LINK\_COUNT / 2){

println("Path is too long, symlink creation failed.")

return

}

val fd = descriptors[fdIdx]

for (step in steps) {

val blockIdx = getFirstFreeBlock() ?: return

fd.addLinkToBlock(blockIdx)

if (step.length > BLOCK\_SIZE) {

println("Name $step is too long, symlink creation failed.")

return

}

val block = dataBlocks[blockIdx]

block.writeData(step)

}

}

private fun createDirectoryLinks(parentIdx: DescriptorIndex, fdIdx: DescriptorIndex) {

val fd = descriptors[fdIdx]

addLinkAtDirectory(fd, "..", parentIdx)

addLinkAtDirectory(fd, ".", fdIdx)

}

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 blocks, operation failed.")

null

} else {

usedBlocks[blockIdx] = true

blockIdx

}

}

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

private fun addLinkAtDirectory(directory: Descriptor, fileName: String, fdIdx: DescriptorIndex) {

if (!directory.isFolder) return

val nameBlock = getFirstFreeBlock() ?: return

val linkBlock = getFirstFreeBlock() ?: return

directory.addLinkToBlock(nameBlock)

directory.addLinkToBlock(linkBlock)

dataBlocks[nameBlock].writeData(fileName)

dataBlocks[linkBlock].data[0] = fdIdx.toByte()

}

private fun getFileName(block: DataBlock): String {return block.data.toString(charset).trim(EMPTY\_CHAR) }

private fun getFdIdx(block: DataBlock): Int {return block.data[0].toInt()}

private fun descrPathLookup(filename: String): DescriptorIndex? {

// returns descriptor of direct parent of file

val lastSep = filename.lastIndexOf(sep)

if (lastSep == -1) {

return currentDirectory

}

return descrLookup(filename.substring(0 until lastSep))

}

private fun descrLookup(name: String, currentDirIdx: DescriptorIndex = currentDirectory, updatePath: Boolean = false): DescriptorIndex? {

// returns descriptor of file

val steps = name.split(sep)

var fdIdx: DescriptorIndex = currentDirIdx

for (step in steps) {

var success = false

val currentDir = descriptors[fdIdx]

for (i in 0 until currentDir.linksToBlocks.size step 2) {

val nameBlock = dataBlocks[currentDir.linksToBlocks[i]]

if (getFileName(nameBlock) == step) {

val linkBlock = dataBlocks[currentDir.linksToBlocks[i+1]]

val parentIdx = fdIdx

fdIdx = getFdIdx(linkBlock)

val fd = descriptors[fdIdx]

if (!fd.isFolder) {

val pathFromLink = getPathFromSymLink(fd)

val idx = descrLookup(pathFromLink, parentIdx, updatePath)

if (idx == null) {

println("$step is not valid link, failed.")

} else {

fdIdx = idx

}

} else if (updatePath) {

if (step == "..") {

path = path.dropLast(1).toMutableList()

} else {

path.add(step)

}

}

success = true

}

}

if (!success) {

println("$step is not a directory or valid symlink.")

return null

}

}

return fdIdx

}

private fun getPathFromSymLink(file: Descriptor): String {

val result = mutableListOf<String>()

for (i in 0 until file.size) {

val step = getFileName(dataBlocks[file.linksToBlocks[i]])

result.add(step)

}

return result.joinToString(sep.toString())

}

private fun getFileNameFromPath(path: String): String {return path.split(sep).last()}

}

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 NO\_SYSTEM\_EXCEPTION = "File system was not mounted."

const val EXT = ".fs"

object Main {

var scan = Scanner(System.`in`)

var fileSystem: FileSystem? = null

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()

println("Unmounted file system to file $fileName.")

} 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) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun create(args: List<String>) {

if (checkArgsCount(args, 1)) {

val fileName = args[0]

fileSystem?.createFile(fileName) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun ls(args: List<String>) {

if (checkArgsCount(args, 0)){

fileSystem?.ls() ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun open(args: List<String>) {

if (checkArgsCount(args, 1)) {

val fileName = args[0]

fileSystem?.openFile(fileName) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun close(args: List<String>) {

if (checkArgsCount(args, 1)) {

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

fileSystem?.closeFile(fdIdx) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

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) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

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) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun link(args: List<String>) {

if (checkArgsCount(args, 2)) {

val fileName = args[0]

val linkName = args[1]

fileSystem?.link(fileName, linkName) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun unlink(args: List<String>) {

if (checkArgsCount(args, 1)) {

val linkName = args[0]

fileSystem?.unlink(linkName) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun truncate(args: List<String>) {

if (checkArgsCount(args, 2)) {

val fileName = args[0]

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

fileSystem?.truncate(fileName, size) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun mkdir(args: List<String>) {

if (checkArgsCount(args, 1)) {

val dirName = args[0]

fileSystem?.mkdir(dirName) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun rmdir(args: List<String>) {

if (checkArgsCount(args, 1)) {

val dirName = args[0]

fileSystem?.rmdir(dirName) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

fun cd(args: List<String>) {

if (args.count() == 1) {

val dirName = args[0]

fileSystem?.cd(dirName) ?: println("NO\_SYSTEM\_EXCEPTION")

} else if (args.isEmpty()) {

fileSystem?.cd() ?: println("NO\_SYSTEM\_EXCEPTION")

} else {

println(WRONG\_ARG\_NUMBER\_EXCEPTION)

}

}

fun symlink(args: List<String>) {

if (checkArgsCount(args, 2)) {

val pathName = args[0]

val linkName = args[1]

fileSystem?.symlink(pathName, linkName) ?: println("NO\_SYSTEM\_EXCEPTION")

}

}

@JvmStatic

fun main(globalArgs: Array<String>) {

while (true) {

if (fileSystem != null) {

val path = fileSystem!!.path.joinToString(sep.toString())

print("$path> ")

}

val input = readLine() ?: ""

val (command, args) = getCommandAndArgs(input)

var exit = false

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)

"mkdir" -> mkdir(args)

"rmdir" -> rmdir(args)

"cd" -> cd(args)

"symlink" -> symlink(args)

"exit" -> {exit = true}

"q" -> {exit = true}

else -> {println("Unknown command")

}

}

} catch (e: IOException) {

e.printStackTrace()

}

if (exit)

break

}

}

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

}

}