МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Лабораторна робота №2

з дисципліни «Системне програмне забезпечення»

на тему *«Файлова система»*

Виконав:

студент гр. ІО-51

Русінов В.В.

Перевірив:

Сімоненко А.В.

Київ 2018

**Завдання**

Разработать драйвер файловой системы для устройства хранения информации блочного типа, используя идею структур данных, описанную выше. В качестве файловой системы взять обычный файл, размер файла определяет объём устройства хранения информации. Самостоятельно выбрать размер блока, количество ссылок на блоки в дескриторе файла, максимальную длину имени файла.

Разработать консольную программу, поддерживающую следующие команды:

* *mount* – подключить файловую систему, сохранённую в файле;
* *umount* – отключить файловую систему, в драйвере не должно остаться каких-либо данных о файловой системе;
* *filestat id* – вывести информацию о дескрипторе файла с указанным номером;
* *ls* – вывести список файлов, с указанием номеров дескрипторов файлов;
* *create имя* – создать файл с заданным именем;
* *open имя* – открыть файл с указанным именем, команда должна назначить уникальный номер *fd*, числовой дескриптор для работы с открытым файлом;
* *close fd* – закрыть ранее открытый файл, уникальный номер *fd* больше не должен быть связан с файлом;
* *read fd смещение размер* – прочитать данные из файла по заданному смещению указанного размера;
* *write fd смещение размер* – записать данные в файл по заданному смещению указанного размера;
* *link имя1 имя2* – создать ссылку с именем *имя2* на существующий файл с именем *имя1*;
* *unlink имя* – уничтожить ссылку с заданным именем;
* *truncate имя размер* – изменить размер файла, если размер файла увеличивается, то неинициализированные данные равны нулю.

**Опис моделі реалізації файлової системи**

Для реалізації файлової системи, було застосовано реалізацію ФС FAT. Для структурування даних було використано таблицю дескрипторів. Дескриптор складається з наступних частин:

* назва файлу;
* розширення файлу;
* атрибути:
  + тільки читання
  + прихований
  + системний
  + архівний
  + ASCII/двійковий
  + тимчасовий
  + блокування запису
* зарезервована частина;
* час створення;
* дата створення;
* вказівник на блок, в який файл записано;
* розмір файлу.

Реалізація директорій зроблена за допомогою таблиці. Кожна директорія може мати “батьківську” та “нащадкову” директорію, тим самим забезпечуючи ієрархію директорій.

Пам’ять надається з урахуванням роботи HDD, тобто ведеться покроковий пошук з початку до кінця.

Символічне лінкування (symlink) не було застосовано в оригінальній реалізації FAT, рішенням цього було створення ще однієї таблиці із кортежами з локаціями двох файлів.

**Програмний код**

Volume.h

#pragma once

class Volume {

private:

char\* \_path;

public:

static int currentdirectory;

static std::list<int> mblocks; // To memory

static std::list<std::tuple<int, int>> linkTable;

static Directory directories[256];

void AddFileToVolume(char\* address);

void InitializeVolume(int size, int blockSize);

void FillInformation(char\* name, char\* ext, short attr, int time, int date, int size);

};

class Driver {

private:

char\* \_path;

int fd;

public:

void mount(char\* path);

void unmount();

void filestat(int id);

void ls();

void create(char\* name, char\* dir = NULL, int size = 0);

bool open(char\* name);

void close();

void read();

void write();

void link(char\* file1, char\* file2);

void unlink(char\* file);

void truncate(char\* file, int size);

long time();

void newdir(char\* name);

};

class Allocator {

private:

public:

static void copycontents(char\* dir);

static int findFreeBlock();

};

class Directory {

private:

char\* \_name;

Directory\* \_child;

int childctr = 0;

Directory\* \_father; /\*only first element should exist\*/

public:

Directory(char\* name);

Directory(char\* name, Directory father);

void setFather(Directory father);

void setChild(Directory child);

char\* getname();

};

Volume.cpp

#include "stdafx.h"

#include "Volume.h"

#include <iostream>

#include <list>

#include <math.h>

#include <ctime>

#include <cwchar>

#include <tuple>

#define BLOCK\_SIZE = 4096; // in bits

void Volume::AddFileToVolume(char \* address)

{

// IO

}

void Volume::InitializeVolume(int size, int blockSize)

{

directories[0] = Directory((char\*)"/"); // mwd

}

void Volume::FillInformation(char \* name, char \* ext, short attr, int time, int date, int size)

{

// attr : ReadOnly/Hidden/System/ASCII-binary/temporary/block

int freeblocknum = Allocator::findFreeBlock();

int information = (int)name << 192 + (int)ext << 168 + attr << 160 + time << 64 + date << 48 + freeblocknum << 32 + size;

Volume::mblocks.push\_back(information);

}

void Driver::mount(char \* path)

{

\_path = path;

Volume::ReadList(path);

Volume::currentdirectory = 0;

}

void Driver::unmount()

{

\_path = NULL;

std::cout << "Volume unmounted, use command mount [volume] to gain access to another volume." << std::endl;

}

void Driver::filestat(int id)

{

int stat = Volume::mblocks[id];

int iname = stat >> 192;

char\* name;

int iext = stat >> 168 | 0xFFFFFFFF000;

char\* ext;

int size = stat & 0xFFFF;

int timedate = (stat >> 80) & 0xFFFF;

int time = timedate & 0xFF00;

int date = timedate & 0x00FF;

sprintf(ext, "%ld", iext);

sprintf(name, "%ld", iname);

printf("File name: %s\nExtension: .%s\nSize: %d bytes\n Time, date: %d, %d.", name, ext, size, time, date);

}

void Driver::ls()

{

std::cout << "Files on the volume:" << std::endl;

for (int i = 0; i < Volume::mblocks.size(); i++) {

int iname = Volume::mblocks[i] >> 192;

char\* name;

sprintf(name, "%ld", iname);

std::cout << name << std::endl;

}

}

void Driver::create(char \* name, char \* dir, int size)

{

if (dir != NULL) {

Allocator::copycontents(dir);

return;

}

if (size != 0) {

int sizeInBlocks = ceil(size / (int)BLOCK\_SIZE);

int\* blockNums = new int[sizeInBlocks];

for (int i = 0; i < sizeInBlocks; i++) {

blockNums[i] = Allocator::findFreeBlock();

if (blockNums[i] == -1) {

std::cout << "Not enough memory" << std::endl;

return;

}

}

long tm = time();

int date = tm >> 16;

int time = tm | 0xFF00;

for (int i = 0; i < sizeInBlocks; i++) {

if (i + 1 == sizeInBlocks) {

Volume::FillInformation(name, "", 0b0000100, time, date, 0xFF, size);

}

if (i + 1 > sizeInBlocks) {

Volume::FillInformation(name, "" /\*\*/, 0b0000100, time, date, -1, size);

continue;

}

Volume::FillInformation(name, "" /\*\*/, 0b0000100, time, date, blockNums[i + 1], size);

}

}

}

bool Driver::open(char \* name)

{

if (fd != NULL) {

std::cout << "You have an opened file" << std::endl;

return false;

}

bool hasdir, copy;

char\*\* dirname;

int j = 0; // char counter

int k = 0; // directory counter

// Directory listed (ex. /folder/folder/file.txt)

for (int i = 0; i < sizeof(name) / sizeof(char); i++) {

if (name[i] == '/')

hasdir = true;

else if (hasdir && name[i] == '/')

copy = true;

else if (copy && name[i] == '/') {

j = 0;

k++;

copy = false;

}

else if (copy) {

dirname[j][k] = name[i];

}

}

for (int d = 0; d < 256; d++) {

bool cont = false;

for (int i = 0; i < k; i++) {

if (Volume::directories[d] == dirname[i]) {

cont = true;

}

}

}

// TODO: DIR hierarchy

// Directory unlisted, only file name given

for (int i = 0; i < Volume::mblocks.size(); i++) {

int iname = Volume::mblocks[i] >> 192;

char\* fname;

sprintf(name, "%ld", iname);

if (fname == name) {

fd = i;

return true;

}

}

std::cout << "File wasn't found." << std::endl;

return false;

}

void Driver::close()

{

fd = NULL;

std::cout << "File closed." << std::endl;

}

void Driver::read()

{

int descr = Volume::mblocks[fd];

// Application can now access the information using 'fd' number;

}

void Driver::write()

{

// Check flags

int fg = (Volume::mblocks[fd] >> 160) & 0xF;

if ((fg & 0b00000001) == 1 /\*write-protected\*/) {

std::cout << "Can't overwrite write-protected files" << std::endl;

return;

}

else if ((fg & 00100000) == 1 /\*system\*/) {

std::cout << "Can't overwrite system files" << std::endl;

return;

}

// Follow the block pointers

int\* ptrs;

bool ptr = true;

ptrs[0] = (Volume::mblocks[fd] >> 32) & 0xFF;

if (ptrs[0] == -1)

ptr = false;

int i = 1;

while (ptr == true) {

ptrs[i] = (Volume::mblocks[ptrs[i - 1]] >> 32) & 0xFF;

if (ptrs[i] == -1)

break;

}

}

void Driver::link(char \* file1, char \* file2)

{

// check if files exist

if (open(file1) != true) {

std::cout << "First file doesn't exist!" << std::endl;

return;

}

if (open(file2) != true) {

std::cout << "Second file doesn't exist!" << std::endl;

}

Volume::linkTable.push\_back(std::make\_tuple(file1, file2));

}

void Driver::unlink(char \* file)

{

for (int i = 0; i < Volume::linkTable.size(); i++) {

if (std::get<0>(Volume::linkTable[i]) == file) {

Volume::linkTable.remove(i);

}

if (std::get<1>(Volume::linkTable[i]) == file) {

Volume::linkTable.remove(i);

}

}

}

int Allocator::findFreeBlock()

{

int block;

for (int i = 0; i < /\*sizeof(Volume::mblocks) / sizeof(int)\*/ Volume::mblocks.size(); i++) {

block = Volume::mblocks[i] | (0xF << 160);

if (block | 0b0 == 0) // isn't write-protected

return i;

}

return -1;

}

void Driver::truncate(char \* file, int size)

{

for (int i = 0; i < Volume::mblocks.size(); i++) {

int iname = Volume::mblocks[i] >> 192;

char\* name;

sprintf(name, "%ld", iname);

if (name == file) {

int initsize = Volume::mblocks[i] & 0xFFFF;

if (initsize > size) {

int sizeInBlocks = ceil(size / (int)BLOCK\_SIZE);

int\* blockNums = new int[sizeInBlocks];

for (int i = 0; i < sizeInBlocks; i++) {

blockNums[i] = Allocator::findFreeBlock();

if (blockNums[i] == -1) {

std::cout << "There's not enough memory. Operation aborted" << std::endl;

return;

}

}

int\* ptrs;

bool ptr = true;

Volume::mblocks[i] &= 0xFF0000;

Volume::mblocks[i] |= blockNums[0] >> 32;

}

}

}

}

long Driver::time()

{

long time = ((pTime.tm\_year - 80) << 25) + (pTime.tm\_mon << 21) +

(pTime.tm\_mday << 16) + (pTime.tm\_hour << 11) +

(pTime.tm\_min << 5) + (pTime.tm\_sec >> 1);

return time;

}

void Driver::newdir(char \* name)

{

for (int i = 0; i < 256; i++) {

if (Volume::directories[i] == NULL) {

Volume::directories[i] = new Directory(name);

}

}

}

Directory::Directory(char \* name)

{

\_name = name;

\_father[0] = Volume::directories[0];

}

Directory::Directory(char \* name, Directory father)

{

\_name = name;

\_father[0] = father;

}

void Directory::setFather(Directory father)

{

\_father[0] = father;

}

void Directory::setChild(Directory child)

{

\_child[childctr] = child;

childctr++;

}

char\* Directory::getname()

{

return \_name;

}

**Висновки**

Під час виконання лабораторної роботи, було ознайомлено із основними принципами роботи файлової системи. Було розроблено програму, що реалізує описану модель файлової системи. Досліджено основні проблеми реалізації файлової системи.