Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут”

Кафедра АСОІУ

**ЗВІТ**

про виконання комп’ютерного практикуму № 4

з дисципліни

“Операційні системи”

|  |  |  |
| --- | --- | --- |
| Прийняв: |  | Виконав: |
| Проф. Сімоненко В. П. |  | студент 3-го курсу  гр. ІП-51 ФІОТ  Булатов Дмитро Єгорович |

Київ – 2017

**ЗМІСТ:**

[1 Опис ідеї реалізації файлової системи 3](#_Toc502014461)

[2 Опис Структур файлової системи 4](#_Toc502014462)

[3 Лістинг програми 5](#_Toc502014463)

[4 Приклад виконання програми 17](#_Toc502014464)

# Опис ідеї реалізації файлової системи

Розроблена файлова систему для пристроїв зберігання інформації блочного типу. Реалізовано однорівневу файлову систему з одного директорією, що містить звичайні файли.

Вміст кожного файлу зберігається в блоках. Для обліку зайнятості блоків підраховується кількість посилань на блок. Блоки в структурах файлової системи адресуються один за одним.

Кожен файл, як об'єкт файлової системи, представлений дескриптором. Кількість дескрипторів заздалегідь задається, тому в файловій системі не може бути створено більше певного кількість файлів, навіть якщо є вільне місце. Дескриптор файлу містить таку інформацію: тип файл, розмір файлу, розташування першого блоку файла.

Файл на диску має структуру зв’язного списку. Блоки данних мают вказівники на наступні, якщо вони існують.

Лінки – це дескриптори файлів що вказують на той самий блок даних. Таким чином при видаленні оригінального файлу лінк продовжить існувати, і стане самостійним файлом. Після створення лінк і оригінальний файл стають рівноправними.

# Опис Структур файлової системи

* ChunkData –заголовок блоку даних.
  + chunkSize – розмір блоку (включає заголовок)
  + nextChunkOffset – положення наступного блоку
  + links – кількість посилань на блок
* FileDescriptor– клас дескриптора файла.
  + Name – ім’я файлу
  + size - розмір файлу
  + offset – посилання на перший блок даних
* FS – клас, реалізуючий файлову систему.
  + FSData – заголовок файлової системи що зберігається на початку файлу
* FSDescriptor – заголовок файлової системи.
  + Size – кількість файлів в FS
  + Descriptors – масив дескрипторів файлів

# Лістинг програми

“Lab4.cpp”

// Lab4.cpp : Defines the entry point for the console application.

//

#include "stdafx.h"

#include <string>

#include <iostream>

#include <Windows.h>

#include <iomanip>

using namespace std;

const size\_t cluster\_size = 512;

struct FileDescriptor {

char name[128];

unsigned long size;

unsigned long offset;

void print() const{

cout << "name: " << name << " size:" << size << " offset:" << offset << "\n";

}

};

struct ChunkData {

unsigned long chunkSize;

unsigned long nextChunkOffset;

unsigned short links;

};

class FS {

private:

struct FSDescriptor

{

unsigned short size;

FileDescriptor Descriptors[2048];

} FSData;

HANDLE FSHandle;

bool opened[2048];

unsigned long DataOffset;

unsigned long DataEnd;

unsigned long FSSize()

{

DWORD A=0, B=0;

B = GetFileSize(FSHandle, &A);

return ((unsigned long)A) << sizeof(DWORD) | B;

}

unsigned long ceil\_cluster(unsigned long a)

{

return a + ((-a) % cluster\_size);

}

void Seek(long offset)

{

SetFilePointer(FSHandle, offset, NULL, FILE\_BEGIN);

}

DWORD ReadData(void\* buffer, unsigned long length, unsigned long offset) {

Seek(offset);

DWORD read;

ReadFile(FSHandle, buffer, length, &read, NULL);

return read;

}

DWORD WriteData(void\* buffer, unsigned long length, unsigned long offset) {

Seek(offset);

DWORD written;

WriteFile(FSHandle, buffer, length, &written, NULL);

return written;

}

void Sync()

{

WriteData(&FSData, sizeof(FSData), 0);

}

void FillZero(unsigned long offset)

{

ChunkData Current = {};

if (!ReadData(&Current, sizeof(ChunkData), offset))

{

cout << "error\n";

}

unsigned char buf[cluster\_size];

memset(buf, 0, cluster\_size);

for (int i=1; i<Current.chunkSize/cluster\_size; i++)

{

WriteData(buf, cluster\_size, offset + cluster\_size \*i);

}

}

unsigned int allocate(unsigned long size)

{

size += (-size) % cluster\_size;

size += cluster\_size;

ChunkData Current;

unsigned long currentOffset = DataOffset;

while (currentOffset<=FSSize())

{

ReadData(&Current, sizeof(ChunkData), currentOffset);

if ( Current.links == 0 )

{

mergeIfPossible(currentOffset);

ReadData(&Current, sizeof(ChunkData), currentOffset);

}

if ( Current.links == 0 && Current.chunkSize >= size )

{

if (Current.chunkSize > size)

{

ChunkData next;

next.links = 0;

next.chunkSize = Current.chunkSize - size;

Current.chunkSize = size;

Current.nextChunkOffset = NULL;

Current.links = 1;

next.nextChunkOffset = NULL;

WriteData(&next, sizeof(ChunkData), currentOffset + size);

}

WriteData(&Current, sizeof(ChunkData), currentOffset);

FillZero(currentOffset);

return currentOffset;

}

else

{

currentOffset += Current.chunkSize;

}

}

Current.chunkSize = size;

Current.links = 1;

Current.nextChunkOffset = NULL;

if (!WriteData(&Current, sizeof(ChunkData), currentOffset))

{

cout << "error\n";

}

FillZero(currentOffset);

return currentOffset;

}

unsigned long getLastChunk(unsigned long offset)

{

while (1)

{

ChunkData Current;

ReadData(&Current, sizeof(ChunkData), offset);

if (!Current.nextChunkOffset) return offset;

offset = Current.nextChunkOffset;

}

}

void incrementLink(unsigned long offset)

{

ChunkData Current;

ReadData(&Current, sizeof(ChunkData), offset);

Current.links++;

WriteData(&Current, sizeof(ChunkData), offset);

}

void mergeIfPossible(unsigned long offset)

{

while(1)

{

ChunkData Cur, Next;

DWORD Read;

ReadData(&Cur, sizeof(ChunkData), offset);

Read = ReadData(&Next, sizeof(ChunkData), offset + Cur.chunkSize);

if (Read && Cur.links == 0 && Next.links == 0)

{

Cur.chunkSize += Next.chunkSize;

WriteData(&Cur, sizeof(ChunkData), offset);

}else

{

return;

}

}

}

void unlink(unsigned long offset)

{

if (!offset) return;

ChunkData Current;

ReadData(&Current, sizeof(ChunkData), offset);

if (Current.links == 0)

{

mergeIfPossible(offset);

return;

}

Current.links -= 1;

WriteData(&Current, sizeof(ChunkData), offset);

if (Current.links == 0)

{

unlink(Current.nextChunkOffset);

}

}

int getFile(const char\* s) const

{

for (int i = 0; i < FSData.size; i++)

{

if (strcmp(FSData.Descriptors[i].name, s) == 0)

return i;

}

return -1;

}

void expand(int FileID, unsigned long new\_size)

{

unsigned long increase = ceil\_cluster(new\_size) - ceil\_cluster(FSData.Descriptors[FileID].size);

FSData.Descriptors[FileID].size = new\_size;

Sync();

if (increase == 0) return;

unsigned long lastBlock = getLastChunk(FSData.Descriptors[FileID].offset);

ChunkData tmp;

ReadData(&tmp, sizeof(ChunkData), lastBlock);

tmp.nextChunkOffset = allocate(increase);

WriteData(&tmp, sizeof(ChunkData), lastBlock);

}

void shrink(int FileID, unsigned long new\_size)

{

unsigned long decrease = ceil\_cluster(FSData.Descriptors[FileID].size) - ceil\_cluster(new\_size);

FSData.Descriptors[FileID].size = new\_size;

Sync();

if (decrease == 0) return;

unsigned long offset = FSData.Descriptors[FileID].offset;

while (1)

{

ChunkData Current;

ReadData(&Current, sizeof(ChunkData), offset);

if (new\_size > (Current.chunkSize - cluster\_size))

{

new\_size -= Current.chunkSize-cluster\_size;

}

else

{

Current.chunkSize = new\_size + cluster\_size;

Current.nextChunkOffset = NULL;

WriteData(&Current, sizeof(ChunkData), offset);

unlink(Current.nextChunkOffset);

return;

}

offset = Current.nextChunkOffset;

}

}

static void PrintByte(unsigned char a)

{

ios::fmtflags f(cout.flags());

cout << "0x" << std::hex << std::uppercase << std::setfill('0') << std::setw(2) << a << " ";

cout.flags(f);

}

unsigned char\* ReadFilePart(unsigned long chunkOffset, unsigned long fileOffset, unsigned long size)

{

unsigned char\* buffer = new unsigned char[size];

unsigned char\* curBuf = buffer;

while (size)

{

ChunkData Current;

ReadData(&Current, sizeof(ChunkData), chunkOffset);

if (Current.chunkSize - cluster\_size <= fileOffset)

{

fileOffset -= Current.chunkSize - cluster\_size;

}

else

{

unsigned long currentRead = min(size - fileOffset, Current.chunkSize - cluster\_size);

ReadData(curBuf, currentRead, chunkOffset + fileOffset + cluster\_size);

size -= currentRead;

fileOffset = 0;

}

}

return buffer;

}

void WriteFilePart(unsigned long chunkOffset, unsigned long fileOffset, unsigned long size, unsigned char\* buffer)

{

unsigned char\* curBuf = buffer;

while (size)

{

ChunkData Current;

ReadData(&Current, sizeof(ChunkData), chunkOffset);

if (Current.chunkSize - cluster\_size <= fileOffset)

{

fileOffset -= Current.chunkSize - cluster\_size;

}

else

{

unsigned long currentRead = min(size - fileOffset, Current.chunkSize - cluster\_size);

WriteData(curBuf, currentRead, chunkOffset + fileOffset + cluster\_size);

size -= currentRead;

fileOffset = 0;

}

fileOffset = Current.nextChunkOffset;

}

}

public:

FS(string path)

{

memset(opened, 0, sizeof(opened));

FSHandle = CreateFileA(path.c\_str(), GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

DataOffset = sizeof(FSDescriptor);

DataOffset += (-DataOffset) % cluster\_size;

if (GetLastError() != ERROR\_ALREADY\_EXISTS)

{

FSData.size = 0;

WriteData(&FSData, sizeof(FSDescriptor), 0);

DataEnd = DataOffset;

}

else

{

ReadData(&FSData, sizeof(FSDescriptor), 0);

DataEnd = FSSize();

}

}

~FS()

{

WriteData(&FSData, sizeof(FSData), 0);

CloseHandle(FSHandle);

}

void ListFiles() const

{

cout << "FS Contains " << FSData.size << " descriptors\n";

for (int i = 0; i < FSData.size; i++)

{

cout << "Descriptor " << i << ": ";

FSData.Descriptors[i].print();

}

}

void OpenFile(const char \*s)

{

int fileIndex = getFile(s);

if (fileIndex == -1)

{

cout << "File not found\n";

return;

}

if (opened[fileIndex])

{

cout << "File already opened\n";

return;

}

opened[fileIndex] = true;

cout << "File " << s << " opened. File ID: " << fileIndex << "\n";

}

void CloseFile(int ID)

{

if (!opened[ID])

{

cout << "File wasn't opened\n";

return;

}

opened[ID] = false;

cout << "File succesfully closed\n";

}

void NewFile(const char\* s)

{

if (getFile(s) != -1)

{

cout << "File already exist\n";

}

int file\_id = FSData.size++;

FSData.Descriptors[file\_id].size = 0;

strcpy(FSData.Descriptors[file\_id].name, s);

FSData.Descriptors[file\_id].offset = allocate(0);

Sync();

cout << "File created succesfully\n";

}

void Link(const char\* s1, const char\* s2)

{

int id1 = getFile(s1), id2 = getFile(s2);

if (id1 == -1)

{

cout << "File " << s1 << " not found\n";

return;

}

if (id2 != -1)

{

cout << "File " << s2 << " exist\n";

return;

}

int file\_id = FSData.size++;

FSData.Descriptors[file\_id].size = FSData.Descriptors[id1].size;

strcpy(FSData.Descriptors[file\_id].name, s2);

FSData.Descriptors[file\_id].offset = FSData.Descriptors[id1].offset;

incrementLink(FSData.Descriptors[id1].offset);

Sync();

cout << "Link created succesfully\n";

}

void truncate(const char\* s, unsigned long new\_size)

{

int id = getFile(s);

if (id == -1)

{

cout << "Incorrect name\n";

return;

}

unsigned long offset = FSData.Descriptors[id].offset;

if (new\_size > FSData.Descriptors[id].size)

{

expand(id, new\_size);

}

else

{

shrink(id, new\_size);

}

for (int i = 0; i < FSData.size; i++)

{

if (i != id && FSData.Descriptors[i].offset == FSData.Descriptors[id].offset)

FSData.Descriptors[i].size = new\_size;

}

cout << "File truncated succesfully\n";

}

void Read(int id, unsigned long offset, unsigned long size)

{

if (!opened[id])

{

cout << "Incorrect ID\n";

return;

}

if (offset + size > FSData.Descriptors->size)

{

cout << "Incorrect offset\n";

return;

}

unsigned char\* buf = ReadFilePart(FSData.Descriptors[id].offset, offset, size);

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

PrintByte(buf[i]);

delete[] buf;

}

void Write(int id, unsigned long offset, unsigned long size, unsigned char\* buf)

{

if (!opened[id])

{

cout << "Incorrect ID\n";

return;

}

if (offset + size > FSData.Descriptors->size)

{

cout << "Incorrect offset\n";

return;

}

WriteFilePart(FSData.Descriptors[id].offset, offset, size, buf);

cout << "Write successful";

}

void RMFile(const char\* s)

{

int id = getFile(s);

if (id == -1)

{

cout << "File don't exist\n";

return;

}

if (opened[id])

{

cout << "File curently opened\n";

return;

}

if (opened[FSData.size - 1])

{

cout << "Can't shift files list. Last file in list opened\n";

return;

}

FileDescriptor tmp = FSData.Descriptors[id];

FSData.Descriptors[id] = FSData.Descriptors[FSData.size - 1];

unlink(tmp.offset);

cout << "File succesfully removed";

}

FileDescriptor\* GetFile(int id) {

if (id >= FSData.size || id < 0) return nullptr;

return FSData.Descriptors + id;

}

} \*CurrentFS;

int main()

{

cout << "Bulatov FS implementation\n";

while (1)

{

string inp;

cout << ">";

cin.clear();

getline(cin, inp);

if (!inp.compare(0,4,"exit"))

{

if (CurrentFS)

delete CurrentFS;

return 0;

}

else

if (!inp.compare(0, 5, "mount"))

{

if (CurrentFS)

{

cout << "FS already mounted\n";

}

else

if (inp.size()<7)

{

cout << "FS location is required\n";

}

else

if (inp[5]!=' ')

{

cout << "unknown command\n";

}

else

{

CurrentFS = new FS(inp.substr(6));

cout << "Succesfully mounted\n";

}

}

else

if (!inp.compare(0, 6, "umount"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

{

delete CurrentFS;

CurrentFS = NULL;

cout << "Succesfully unmounted\n";

}

}

else

if (!inp.compare(0, 4, "open"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<6)

{

cout << "File name is required\n";

}

else

{

CurrentFS->OpenFile(inp.substr(5).c\_str());

}

}

else

if (!inp.compare(0, 5, "close"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<7)

{

cout << "File id is required\n";

}

else

{

CurrentFS->CloseFile(atoi(inp.substr(6).c\_str()));

}

}

else

if (!inp.compare(0, 6, "create"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<8)

{

cout << "File name is required\n";

}

else

{

CurrentFS->NewFile(inp.substr(7).c\_str());

}

}

else

if (!inp.compare(0, 7, "unlink"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<9)

{

cout << "File name is required\n";

}

else

{

CurrentFS->RMFile(inp.substr(8).c\_str());

}

}

else

if (!inp.compare(0, 4, "link"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<6)

{

cout << "File names is required\n";

}

else

{

string names = inp.substr(5);

string name1 = names.substr(0, names.find(' '));

string name2 = names.substr(names.find(' ') + 1);

CurrentFS->Link(name1.c\_str(), name2.c\_str());

}

}

else

if (!inp.compare(0, 8, "truncate"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<10)

{

cout << "File names is required\n";

}

else

{

string params = inp.substr(9);

string name1 = params.substr(0, params.find(' '));

string new\_size = params.substr(params.find(' ') + 1);

CurrentFS->truncate(name1.c\_str(), atoi(new\_size.c\_str()));

}

}

else

if (!inp.compare(0, 4, "read"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<6)

{

cout << "File names is required\n";

}

else

{

string params = inp.substr(5);

string file\_id = params.substr(0, params.find(' '));

params = params.substr(params.find(' ') + 1);

string offset = params.substr(0, params.find(' '));

string size = params.substr(params.find(' ') + 1);

CurrentFS->Read(atoi(file\_id.c\_str()), atoi(offset.c\_str()), atoi(size.c\_str()));

}

}

else

if (!inp.compare(0, 5, "write"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<7)

{

cout << "File names is required\n";

}

else

{

string params = inp.substr(6);

string file\_id = params.substr(0, params.find(' '));

params = params.substr(params.find(' ') + 1);

string offset = params.substr(0, params.find(' '));

int size = atoi(params.substr(params.find(' ') + 1).c\_str());

unsigned char\* buffer = new unsigned char[size];

cout << "Enter data to write:";

cin >> buffer;

CurrentFS->Write(atoi(file\_id.c\_str()), atoi(offset.c\_str()), size, buffer);

delete[] buffer;

}

}

else

if (!inp.compare(0, 2, "ls"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

{

CurrentFS->ListFiles();

}

}

else

if (!inp.compare(0, 8, "filestat"))

{

if (!CurrentFS)

{

cout << "FS not mounted\n";

}

else

if (inp.size()<10)

{

cout << "File id is required\n";

}

else

{

FileDescriptor\* c = CurrentFS->GetFile(atoi(inp.substr(9).c\_str()));

if (c)

c->print();

else

cout << "Bad id\n";

}

}

else

{

cout << "unknown command\n";

}

}

return 0;

}

# Приклад виконання програми

