МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО» ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Курсова робота

по курсу

«Системне програмне забезпечення»

Виконала:

Студентка ІО-з91

Карнаухова Анастасія Олександрівна

Перевірив:

Сімоненко Андрій Валерійович

Киев – 2022

Тема:

Файлова система.

Опис форматів даних Файлової Системи:

Кожен файл, як об’єкт ФС, представлений дескриптором файлу. Кількість дескрипторів файлів заздалегідь задається, тому у ФС не може бути створено більше певної кількості файлів, навіть якщо в носії інформації є вільне місце.

Всі дескриптори файлів зберігаються у масиві, що дозволяє визначити позицію дескриптора файлу на носії інформації за його номером. Дескриптор файлу повинен містити, принаймні, таку інформацію: тип файлу (звичайний файл, символічне посилання або директорія), кількість жорстких посилань на файл, розмір файлу в байтах, мапу номерів блоків файлу.

Мапа номерів блоків має наступну структуру: у дескрипторі файлу є кілька прямих посилань на блоки та є одне посилання на блок, який містить прямі посилання на блоки (посилання на блок – це номер блоку). Позиція посилання в мапі визначає зміщення даних у файлі, посилання нумеруються підряд. Посилання не вказує на блок у двох випадках: відповідне зміщення перевищує розмір файлу або значення всіх байтів у відповідному блоці дорівнюють нулю.

Звичайний файл – це тип файлу, який використовуються для зберігання будь-яких даних. Драйвер ФС не інтерпретує його вміст, ядро не інтерпретує вміст довільного звичайного файлу. Директорія – це тип файлу, вміст якого складається із записів директорії (directory entry). Формат запису директорії визначається форматом ФС та не інтерпретується тільки драйвером ФС. Запис директорії називається «жорстким посиланням». Запис директорії складається з імені файлу та порядкового номеру файлу. Максимальна довжина імені файлу та дозволені символи в імені визначаються форматом ФС.

Порядковий номер файлу – це унікальний номер файлу в межах ФС, де він зберігається, у курсовій роботі це значення дорівнює номеру дескриптора файлу.

Оскільки ім’я файлу не є частиною дескриптора файлу, тому кілька жорстких посилань можуть вказувати на один файл, тобто один файл може мати кілька імен. Оскільки жорсткі посилання на файли можуть бути знищені, необхідно передбачити можливість позначити недійсні жорсткі посилання на файли в директорії.

Дерево директорій дозволяє організовувати жорсткі посилання на файли в ієрархічну структуру. Дерево директорій будується за допомогою створення жорстких посилань на директорії.

Дескриптор файлу кореневої директорії / (root directory) або початкової директорії дерева ФС має фіксований номер, який визначається форматом ФС. Дескриптори інших директорій мають довільні номери.

Жорсткі посилання на файли різних типів мають однаковий формат у директорії. Будь-яка директорія завжди має два наперед визначених жорсткі посилання з іменами . (dot) та .. (dot-dot), які вказують на поточну директорію та батьківську директорію

відповідно. Жорстке посилання з ім’ям .. у кореневій директорії вказує на кореневу директорію.

Шляхове ім’я (pathname) може бути абсолютним (absolute) або відносним (relative). Абсолютне шляхове ім’я починається з імені кореневої директорії /. Відносне шляхове ім’я не починається з імені кореневої директорії.

Пошук файлу за абсолютним шляховим іменем (file name lookup) починається з кореневої директорії. Пошук файлу за відносним шляховим іменем починається з поточної робочої директорії (CWD, current working directory) процесу. Кожен процес має CWD, яку він може змінювати. Кожний компонент шляхового імені після розкриття символічних посилань, за винятком останнього компонента, повинен бути жорстким посиланням на директорію.

При виконанні пошуку файлу за шляховим ім’ям необхідно враховувати такі компоненти шляхового імені як ., .. та символічні посилання, кількість та використання цих компонентів у шляховому імені можуть бути довільними.

Символічне посилання (symlink, symbolic link) – це тип файлу, вміст якого – це рядок. При створенні символічного посилання вміст рядка не інтерпретується ані ядром, ані драйвером ФС. Символічне посилання є ще одним типом файлу, на додачу до звичайного файлу та директорії.

Якщо деякий компонент шляхового імені є символічним посиланням, тоді вміст символічного посилання необхідно додати замість жорсткого посилання на символічне посилання у шляховий файл та відкинути ті компоненти шляхового імені, які вже пройдені. Інакше кажучи, вміст символічного посилання треба конкатенувати з рештою компонентів шляхового імені.

Після додавання вмісту символічного посилання, необхідно почати пошук файлу з першого компоненту отриманого шляхового імені. Тобто символічне посилання може створювати абсолютне або відносне шляхове ім’я. Якщо вміст символічного посилання починається з символу ‘/’, тоді пошук файлу продовжується починаючи з кореневої директорії, інакше пошук файлу продовжується починаючи з директорії в якій знаходиться жорстке посилання на символічне посилання.

Символічне посилання може «вказувати» на файл, але воно не є ще одним жорстким посиланням на файл (ще одним ім’ям файлу). Тому створення та знищення символічного посилання не призводить до зміни значення лічильника жорстких посилань у дескрипторі файлу, на який «вказує» символічне посилання. Тому можна створювати або може бути символічне посилання яке «вказує» на файл, який не існує у ФС.

Символічні посилання можуть «вказувати» на інші символічні посилання та на директорії вище по дереву ФС. Тому можливо створити цикл у шляховому імені використовуючи одне або кілька символічних посилань. Для запобігання цього циклу при пошуку файлу за шляховим ім’ям необхідно задати максимальну можливу кількість переходів по символічним посиланням.

Лістинг програми:

Dir.cpp

```
#include "Dir.h"


Dir::Dir(const string name) {
  this->name = name;
  this->size = 0;
  this->type = 1;
  this->firstChild = NULL;
  this->lastChild = NULL;
  this->parentNode = NULL;
}


void Dir::setParent(Dir* parentNode) {
  this->parentNode = parentNode;
}


string Dir::getAbsolutePath() {
  string path = "";
  if (this == this->parent()) {
    return "/";
  }
  for (auto p = this; p != p->parent(); p = p->parent()) {
    path = "/"+p->getName()+path;
  }
  return path;
}


Dir* Dir::parent() {
```

```
  return this->parentNode;
}

File* Dir::first() {
  return this->firstChild;
}

File* Dir::last() {
  return this->lastChild;
}

File* Dir::getElementByName(const string name) {
  for (auto p = firstChild; p != NULL; p = p->next()) {
    if (p->getName() == name) {
      return p;
    }
  }
  return NULL;
}

File* Dir::appendChild(File* pNewNode) {
  pNewNode->setNext(NULL);
  pNewNode->setPrev(this->lastChild);
  if (pNewNode->isDir()) {
    static_cast<Dir*>(pNewNode)->parentNode = this;
  }
  if (this->lastChild != NULL) {
 this->lastChild->setNext(pNewNode);
  } else {
    this->firstChild = pNewNode;
  }
```

```
  this->lastChild = pNewNode;
  this->size++;
  return pNewNode;
}

void Dir::removeChild(File* pNode) {
  if (pNode == firstChild) {
    if (pNode == lastChild) {
      firstChild = NULL;
      lastChild = NULL;
    } else {
      pNode->next()->setPrev(pNode->prev());
      firstChild = pNode->next();
    }
  } else if (pNode == lastChild) {
    pNode->prev()->setNext(pNode->next());
    lastChild = pNode->prev();
  } else {
    pNode->prev()->setNext(pNode->next());
    pNode->next()->setPrev(pNode->prev());
  }
  this->size--;
  if (pNode->isDir()) {
    Dir* dir = static_cast<Dir*>(pNode);
    dir->forEach([&dir](File* p) -> void {
      dir->removeChild(p);
    });
  }

  delete pNode;
}
```

```
void Dir::forEach(function<void(File*)> callback) {
  for (auto p = firstChild; p != NULL; p = p->next()) {
    callback(p);
  }
}

Dir* Dir::cloneNode() {
  Dir* newNode = new Dir(this->name);
  this->forEach([&newNode](File* p) -> void {
    newNode->appendChild(p->cloneNode());
  });
  return newNode;
}

bool Dir::contains(Dir* pNode) {
  while (pNode != pNode->parent()) {
    if (pNode == this) {
      return true;
    }
    pNode = pNode->parent();
  }
  return false;
}
```

Dir.h

```
#pragma once
#include <functional>
#include "File.h"

class Dir:public File {
  private:
```

```
    File* firstChild;
    File* lastChild;
    Dir* parentNode;

  public:
    Dir(const string);
    ~Dir();
    string getAbsolutePath();
    File* appendChild(File*);
    void removeChild(File*);
    void setParent(Dir*);
    virtual Dir* cloneNode();
    bool contains(Dir*);
    Dir* parent();
    File* first();
    File* last();
    void forEach(function<void(File*)>);
    File* getElementByName(const string);
};
```

File.cpp

```
#include "File.h"
File::File() {
}
File::File(const string name, const string content) {
  this->name = name;
  this->content = content;
  this->size = content.length();
  this->type = 0;
}

void File::setNext(File* pNextNode) {
```

```
  this->nextSibling = pNextNode;
}

void File::setPrev(File* pPrevNode) {
  this->previousSibling = pPrevNode;
}

string File::getName() {
  return this->name;
}

void File::setName(const string name) {
  this->name = name;
}

int File::getSize() {
  return this->size;
}

string File::getContent() {
  return this->content;
}

void File::setContent(const string& content) {
  this->content = content;
}

File* File::next() {
  return this->nextSibling;
}
```

```
File* File::prev() {
  return this->previousSibling;
}

bool File::isDir() {
  if (this->type == 1) {
    return true;
  }
  return false;
}

File* File::cloneNode() {
  File* newNode = new File(this->name, this->content);
  return newNode;
}

File::~File() {

}
```

File.h

```
#pragma once
#include <string>
#include <iostream>
#include <cstdio>
using namespace std;

class File {
  private:
```

```
    string content;
  protected:
    string name;
    int size;
    int type;
    File* nextSibling;
    File* previousSibling;

  public:
    File();
    File(const string, const string);
    ~File();
    string getName();
    void setName(const string);
    int getSize();
    string getContent();
    void setContent(const string&);
    File* next();
    File* prev();
    virtual File* cloneNode();
    void setNext(File*);
    void setPrev(File*);
    bool isDir();
};
```

FileSystem.cpp

```
#include "FileSystem.h"
```

```
FileSystem::FileSystem() {

  this->rootDir = new Dir("/");

  rootDir->setParent(rootDir);

  this->currentDir = this->rootDir;

    helpMessage = "usage: <command> [path1] [path2]\n\nThe most commonly used commands are:\ncd Switch to some directory\ncp         Copy and paste\nexit     Back to terminal\nhelp     Display this guide\nls List directory's content\nmkdir    Create a directory\nmv       Move/Rename a file or directory\nrm       Remove a file or directory\ntouch   Create a file\n\n\"help <command>\" show specifical guides\n";

}


FileSystem::~FileSystem() {



}


Dir* FileSystem::getRootDir() {

  return this->rootDir;

}


Dir* FileSystem::getCurrentDir() {

  return this->currentDir;

}


void FileSystem::route(const string& cmd) {

  if (cmd.empty()) {

    return;

  }


  auto elems = split(cmd, ' ');

  int length = elems.size();

  auto op = elems[0];


  if (op == "cd") {

    if (length > 2) {
```

```
    cout<<"Invalid arguments!!!"<<endl;
  } else if (length == 2) {
    switchToDir(elems[1]);
  }
} else if (op == "ls") {
  if (length > 2) {
    cout<<"Invalid arguments!!!"<<endl;
  } else if (length == 1) {
    listDir("");
  } else {
    listDir(elems[1]);
  }
} else if (op == "touch") {
  if (length != 2) {
    cout<<"Invalid arguments!!!"<<endl;
  } else {
    makeFile(elems[1]);
  }
} else if (op == "mkdir") {
  if (length != 2) {
    cout<<"Invalid arguments!!!"<<endl;
  } else {
    makeDir(elems[1]);
  }
} else if (op == "rm") {
  if (length != 2) {
    cout<<"Invalid arguments!!!"<<endl;
  } else {
    remove(elems[1]);
  }
} else if (op == "cp") {
```

```
  if (length != 3) {
    cout<<"Invalid arguments!!!"<<endl;
  } else {
    copy(elems[1], elems[2]);
  }
} else if (op == "mv") {
  if (length != 3) {
    cout<<"Invalid arguments!!!"<<endl;
  } else {
    move(elems[1], elems[2]);
  }
} else if (op == "exit") {
  exit(0);
} else if (op == "help") {
  if (length == 1) {
    cout<<helpMessage<<endl;
  } else if (length == 2) {
    if (elems[1] == "cd") {
      cout<<"cd [path]"<<endl;
    } else if (elems[1] == "cp") {
      cout<<"cp [source] [destination]"<<endl;
    }  else if (elems[1] == "ls") {
      cout<<"ls [path]"<<endl;
    } else if (elems[1] == "mv") {
      cout<<"mv [source] [destination]"<<endl;
    } else if (elems[1] == "mkdir") {
      cout<<"mkdir [directory name]"<<endl;
    } else if (elems[1] == "rm") {
      cout<<"rm [target]"<<endl;
    } else if (elems[1] == "touch") {
      cout<<"touch [filename]"<<endl;
```

```
      } else {
        cout<<"No help for "<<elems[1]<<endl;
      }
    } else {
      cout<<"Invalid arguments!!!"<<endl;
    }
  } else {
    cout<<"Invalid command!!!"<<endl;
  }

  return;
}

const vector<string> FileSystem::split(const string& s, const char& delimiter) {
  stringstream ss(s);
  vector<string> elems;
  string item;
  while (getline(ss, item, delimiter)) {
    elems.push_back(item);
  }

  return elems;
}

const vector<string> FileSystem::parse(const string& path) {
  return split(path, '/');
}

File* FileSystem::getElementByPath(const vector<string>& elems) {
  Dir* p = NULL;
  File* elem = NULL;
```

```
int i = 0;


if (elems.empty()) {
  return this->currentDir;
}


if (elems[0] == "") { // Absolute path
  p = this->rootDir;
  i = 1;
} else { // Relative path
  p = this->currentDir;
  i = 0;
}


for (; i < elems.size(); i++) {
  if (elems[i] == "..") {
    p = p->parent();
    continue;
  } else {
    elem = p->getElementByName(elems[i]);
    if (elem != NULL) {
      if (elem->isDir()) {
        if (i == elems.size() - 1) {
          return elem;
        } else {
          p = static_cast<Dir*>(elem);
          continue;
        }
      } else {
        if (i == elems.size() - 1) {
          return elem;
```

```
        } else {
          printf("%s is a file !!!\n", elems[i].c_str());
          break;
        }
       }
     } else {
       return NULL;
     }
   }
  }

  return p;
}

Dir* FileSystem::getParentDirByPath(const vector<string>& path, function<void(const string&, Dir*)> callback) {
  vector<string> tmp(path);
  Dir* parentDir;
  auto name= *tmp.rbegin();
  tmp.pop_back();

  if (name == "") {
    printf("Dir or file name should not be empty!!!\n");
  } else {
    parentDir = static_cast<Dir*>(getElementByPath(tmp));
    if (parentDir != NULL) {
      callback(name, parentDir);
    } else {
      printf("Path is not valid!!!\n");
    }
  }
```

```
  return parentDir;
}

void FileSystem::switchToDir(const string& path) {
  auto elems = parse(path);
  auto elem = getElementByPath(elems);
  if (elem != NULL) {
    if (elem->isDir()) {
      this->currentDir = static_cast<Dir*>(elem);
    } else {
      printf("%s is a file!!!\n", (*elems.rbegin()).c_str());
    }
  } else {
    printf("Path is not valid!!! %s\n", path.c_str());
  }
}

void FileSystem::listDir(const string& path) {
  Dir* dir = getCurrentDir();
  if (!path.empty()) {
    auto elem = getElementByPath(parse(path));
    if (elem != NULL) {
      dir = static_cast<Dir*>(elem);
    }
  }
  if (dir->getSize() != 0) {
    dir->forEach([](File* p) -> void {
      if (p->isDir()) {
        printf("\033[01;34m%s\033[0m  ", p->getName().c_str());
      } else {
        printf("%s  ", p->getName().c_str());
```

```
    }
   });
   printf("\n");
  }
}

Dir* FileSystem::makeDir(const string& path) {
  if (path.empty()) {
   printf("Dir name should not be empty\n");
   return NULL;
  }

  Dir* dir = NULL;

  getParentDirByPath(parse(path), [&dir](const string& dirName, Dir* parentDir) -> void {
   if (parentDir->getElementByName(dirName) == NULL) {
    dir = new Dir(dirName);
    parentDir->appendChild(dir);
   } else {
    printf("%s exists!!!\n", dirName.c_str());
   }
  });

  return dir;
}

File* FileSystem::makeFile(const string& path) {
  if (path.empty()) {
   printf("File name should not be empty\n");
   return NULL;
  }
```

```
  File* file = NULL;

  getParentDirByPath(parse(path), [&file](const string& fileName, Dir* parentDir) -> void {
    if (parentDir->getElementByName(fileName) == NULL) {
      file = new File(fileName, "");
      parentDir->appendChild(file);
    } else {
      printf("%s exists!!!\n", fileName.c_str());
    }
  });

  return file;
}

File* FileSystem::copy(const string& srcPath, const string& dstPath) {
  if (srcPath.empty() || dstPath.empty()) {
    printf("Dir or file name should not be empty\n");
    return NULL;
  }

  File* node = NULL;
  auto src = getElementByPath(parse(srcPath));
  if (src != NULL) {
    getParentDirByPath(parse(dstPath), [&node, src, srcPath, dstPath](const string& dstName, Dir* parentDir)
-> void {
      if (parentDir->getElementByName(dstName) == NULL) {
        if (src->isDir() && static_cast<Dir*>(src)->contains(parentDir)) {
          printf("%s is child of %s\n", dstPath.c_str(), srcPath.c_str());
        } else {
          node = src->cloneNode();
          node->setName(dstName);
```

```
        parentDir->appendChild(node);
       }
     } else {
       printf("%s exists!!!\n", dstName.c_str());
     }
   });
  } else {
   printf("%s not exists!!!\n", srcPath.c_str());
  }

  return node;
}

void FileSystem::remove(const string& path) {
  if (path.empty()) {
   printf("Dir or file name should not be empty\n");
   return;
  }

  getParentDirByPath(parse(path), [this](const string& name, Dir* parentDir) -> void {
   auto elem = parentDir->getElementByName(name);
   if (elem != NULL) {
     if (elem->isDir() && static_cast<Dir*>(elem)->contains(this->currentDir)) {
       printf("Can't remove parent directory!!!\n");
     } else {
       parentDir->removeChild(elem);
     }
   } else {
     printf("%s not exists!!!\n", name.c_str());
   }
  });
```

```
}

File* FileSystem::move(const string& srcPath, const string& dstPath) {
  auto dst = copy(srcPath, dstPath);
  if (dst != NULL) {
    remove(srcPath);
  }
  return dst;
}

void FileSystem::readFile(const string& filePath) {
  auto file = getElementByPath(parse(filePath));
  if (file == NULL) {
    printf("%s not exists!!!\n", filePath.c_str());
  } else if (file->isDir()) {
    printf("%s is a directory!!!\n", filePath.c_str());
  } else {
    cout<<file->getContent()<<endl;
  }
}

void FileSystem::writeFile(const string& filePath, const string& content) {
  auto file = getElementByPath(parse(filePath));
  if (file == NULL) {
    printf("%s not exists!!!\n", filePath.c_str());
  } else if (file->isDir()) {
    printf("%s is a directory!!!\n", filePath.c_str());
  } else {
    file->setContent(content);
  }
}
```

FileSystem.h

```
#pragma once
#include <vector>
#include <sstream>
#include "File.h"
#include "Dir.h"

class FileSystem {
  private:
    Dir* rootDir;
    Dir* currentDir;

  public:
    FileSystem();
    ~FileSystem();

    string helpMessage;
    void route(const string&);
    static const vector<string> split(const string&, const char&);
    static const vector<string> parse(const string&);
    Dir* getCurrentDir();
    Dir* getRootDir();
    File* getElementByPath(const vector<string>&);
    Dir* getParentDirByPath(const vector<string>&, function<void(const string&, Dir*)>);

    void switchToDir(const string&);
    void listDir(const string&);
    Dir* makeDir(const string&);

    File* makeFile(const string&);
    void writeFile(const string&, const string&);
```

```
    void readFile(const string&);


    File* copy(const string&, const string&);
    void remove(const string&);
    File* move(const string&, const string&);
};
```

Main.cpp

```
#include "FileSystem.h"


int main () {
  system("clear");
  auto fs = new FileSystem();
  cout<<fs->helpMessage<<endl;
  string cmd = "";
  cout<<"mS@MS:"<<fs->getCurrentDir()->getAbsolutePath()<<"$ ";
  while(getline(cin, cmd)) {
    fs->route(cmd);
    cout<<"mS@MS:"<<fs->getCurrentDir()->getAbsolutePath()<<"$ ";
  }
  return 0;
}
```

Test.cpp

```
#define CATCH_CONFIG_MAIN
#include <unordered_map>
#include "FileSystem.h"
#include "../lib/catch.hpp"


SCENARIO ("Parse a path", "") {
  GIVEN ("A path") {
    string path = "/home/ziqi/FileSystem/src/Dir.c/";
    WHEN ("Path parsed") {
```

```
    auto elems = FileSystem::parse(path);
    REQUIRE(elems.size() == 6);
    REQUIRE(elems[0] == "");
    REQUIRE(elems[1] == "home");
    REQUIRE(elems[2] == "ziqi");
    REQUIRE(elems[3] == "FileSystem");
    REQUIRE(elems[4] == "src");
    REQUIRE(elems[5] == "Dir.c");
    //REQUIRE(elems[6] == "");
   }
  }
}

SCENARIO ("Dir test", "") {
  GIVEN ("An instance of Dir") {
   auto dir = new Dir("/");
   unordered_map<string, bool> m;
   WHEN ("Append some children") {
    auto a_txt = dir->appendChild(new File("a.txt", "Hello, world"));
    auto src = dir->appendChild(new Dir("src"));
    auto out = dir->appendChild(new Dir("out"));
    auto b_out = dir->appendChild(new File("b.out", ""));

    REQUIRE(dir->getSize() == 4);

    dir->forEach([&m](File* p) -> void {
      m[p->getName()] = p->isDir();
    });
    REQUIRE(m["a.txt"] == 0);
    REQUIRE(m["src"] == 1);
    REQUIRE(m["out"] == 1);
```

```
    REQUIRE(m["b.out"] == 0);


    THEN ("Find dir or file by name") {
      auto elem = dir->getElementByName("a.txt");
      REQUIRE(elem->getName() == "a.txt");
      REQUIRE(elem->getContent() == "Hello, world");
      elem = dir->getElementByName("src");
      REQUIRE(elem->getName() == "src");
      elem = dir->getElementByName("out");
      REQUIRE(elem->getName() == "out");
      elem = dir->getElementByName("b.out");
      REQUIRE(elem->getName() == "b.out");
      REQUIRE(elem->getContent() == "");
    }


    THEN ("Remove children") {
      dir->removeChild(src);
      dir->removeChild(b_out);
      REQUIRE(dir->getSize() == 2);
      m.clear();
      dir->forEach([&m](File* p) -> void {
        m[p->getName()] = p->isDir();
      });
      REQUIRE(m["a.txt"] == 0);
      REQUIRE(m["out"] == 1);
      REQUIRE(m.find("src") == m.end());
      REQUIRE(m.find("b.out") == m.end());
    }
  }
 }
}
```

```
SCENARIO ("mkdir", "") {
  auto fs = new FileSystem();
  unordered_map<string, bool> m;
  WHEN ("Created at current dir") {
    fs->makeDir("src");
    fs->makeDir("out");
    REQUIRE(fs->getCurrentDir()->getSize() == 2);
    fs->getCurrentDir()->forEach([&m](File* p) -> void {
      m[p->getName()] = p->isDir();
    });
    REQUIRE(m["src"] == 1);
    REQUIRE(m["out"] == 1);
  }
  WHEN ("Created by absolute dir") {
    REQUIRE(fs->makeDir("src/headers") == NULL);
    auto src = fs->makeDir("/src/");
    REQUIRE(src != NULL);
    REQUIRE(src->getName() == "src");
    auto headers = fs->makeDir("/src/headers/");
    REQUIRE(headers != NULL);
    REQUIRE(headers->getName() == "headers");
    src->forEach([&m](File* p) -> void {
      m[p->getName()] = p->isDir();
    });
    REQUIRE(m["headers"] == 1);
  }
}

SCENARIO ("Switch to some dir", "mkdir") {
  GIVEN ("Dirs") {
```

```
    auto fs = new FileSystem();
    auto src = fs->makeDir("src");
    auto out = fs->makeDir("src/out/");
    WHEN ("Switch to relative path") {
      fs->switchToDir("src");
      REQUIRE(fs->getCurrentDir() == src);
      fs->switchToDir("out");
      REQUIRE(fs->getCurrentDir() == out);
      THEN ("Switch to parent") {
        fs->switchToDir("..");
        REQUIRE(fs->getCurrentDir() == src);
        fs->switchToDir("../");
        REQUIRE(fs->getCurrentDir() == fs->getRootDir());
      }
    }
    WHEN ("Switch to absolute path") {
      fs->switchToDir("/src/out/");
      REQUIRE(fs->getCurrentDir() == out);
      fs->switchToDir("..");
      fs->switchToDir("/src/out");
      REQUIRE(fs->getCurrentDir() == out);
    }
  }
}

SCENARIO ("touch" , "mkdir") {
  auto fs = new FileSystem();
  unordered_map<string, bool> m;

  WHEN ("Created by relative path") {
    fs->makeFile("MakeFile");
```

```
    fs->makeDir("src");
    fs->getCurrentDir()->forEach([&m](File* p) -> void {
      m[p->getName()] = p->isDir();
    });
    REQUIRE(fs->getCurrentDir()->getSize() == 2);
    REQUIRE(m["MakeFile"] == 0);
    REQUIRE(m["src"] == 1);
  }

  WHEN ("Created by absolute path") {
    fs->makeDir("/src");
    fs->makeFile("/src/File.c");
    fs->switchToDir("/src/");
    REQUIRE(fs->getCurrentDir()->getSize() == 1);
    fs->getCurrentDir()->forEach([&m](File* p) -> void {
      m[p->getName()] = p->isDir();
    });
    REQUIRE(m["File.c"] == 0);
  }
}

SCENARIO ("rm cp", "mkdir touch") {
  auto fs = new FileSystem();
  unordered_map<string, bool> m;
  auto src = fs->makeDir("src");
  fs->makeDir("src/headers");
  fs->makeFile("src/headers/File.h");
  fs->makeFile("src/File.c");
  auto out = fs->makeDir("out");

  WHEN ("Remove file or dir") {
```

```
  fs->switchToDir("src");
  fs->remove("File.c");
  REQUIRE(fs->getCurrentDir()->getSize() == 1);
  fs->getCurrentDir()->forEach([&m](File* p) -> void {
    m[p->getName()] = p->isDir();
  });
  REQUIRE(m["headers"] == 1);
  REQUIRE(m.find("File.c") == m.end());
  fs->switchToDir("..");
  fs->remove("src");
  REQUIRE(fs->getCurrentDir()->getSize() == 1);
  m.clear();
  fs->getCurrentDir()->forEach([&m](File* p) -> void {
    m[p->getName()] = p->isDir();
  });
  REQUIRE(m.find("src") == m.end());
  REQUIRE(m["out"] == 1);
}

WHEN ("Copy file or dir") {
  fs->copy("/src/headers/", "/out/headers/");
  fs->switchToDir("out");
  REQUIRE(out->getSize() == 1);
  fs->getCurrentDir()->forEach([&m](File* p) -> void {
    m[p->getName()] = p->isDir();
  });
  REQUIRE(m["headers"] == 1);
  fs->switchToDir("headers");
  m.clear();
  fs->getCurrentDir()->forEach([&m](File* p) -> void {
    m[p->getName()] = p->isDir();
```

```
    });
    REQUIRE(m["File.c"] == 0);
  }
}
```

Приклади виконання запитів:

```
usage: <command> [path1] [path2]


The most commonly used commands are:

cd      Switch to some directory

cp      Copy and paste

exit    Back to terminal

help    Display this guide

ls      List directory's content

mkdir   Create a directory

mv      Move/Rename a file or directory

rm      Remove a file or directory

touch   Create a file


"help <command>" show specifical guides


anastasia@Kr:/$ mkdir test

anastasia@Kr:/$ ls

test

anastasia@Kr:/$ touch file1

anastasia@Kr:/$ ls

test  file1

anastasia@Kr:/$ cd test

anastasia@Kr:/test$ touch f1

anastasia@Kr:/test$ touch f2

anastasia@Kr:/test$ touch f3

anastasia@Kr:/test$ ls

f1  f2  f3

anastasia@Kr:/test$ cd
```

```
anastasia@Kr:/test$ cd ..

anastasia@Kr:/$ ls

test  file1

anastasia@Kr:/$ ls test

f1  f2  f3

anastasia@Kr:/$ rm test/f3

anastasia@Kr:/$ ls test

f1  f2

anastasia@Kr:/$
```