

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ УКРАИНЫ

«КИЕВСКИЙ ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ»

ФАКУЛЬТЕТ ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

ЛАБОРАТОРНАЯ РАБОТА №3

ПО КУРСУ

«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ»

Выполнил:

студент IV курса

группы ІО-21

Журо Георгий Александрович

Киев – 2015

**Тема**: Файловая система.

**Задание:**

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.

Добавить в драйвер файловой системы, разработанный в лабораторной работе №3, поддержку дерева директорий и символических ссылок. Добавить в консольную программу команды для работы с директориями и символическими ссылками.

Консольная программа должна поддерживать путевые имена файлов в командах, файл ищется по путевому имени начиная с текущей рабочей директории. Команды *create*, *open*, *truncate* и *cd* должны работать с файлами на которое указывает имя (если последняя компонента имени файла это символическая ссылка, то необходимо перейти по этой ссылке). Все остальные команды, которые работают с именами файлов, работают с заданными файлами по имени (если последняя компонента имени файла это символическая ссылка, то команда применяется к самой символической ссылки, а не к файлу на который указывает символическая ссылка). Команда *link* должна создавать жёсткие ссылки только для обычных файлов.

Разработать консольную программу, поддерживающую следующие команды:

1. mount – подключить файловую систему, сохранённую в файле;
2. umount – отключить файловую систему, в драйвере не должно остаться каких-либо данных о файловой системе;
3. filestat id – вывести информацию о дескрипторе файла с указанным номером;
4. ls – вывести список файлов, с указанием номеров дескрипторов файлов;
5. create имя – создать файл с заданным именем;
6. open имя – открыть файл с указанным именем, команда должна назначить уникальный номер fd, числовой дескриптор для работы с открытым файлом;
7. close fd – закрыть ранее открытый файл, уникальный номер fd больше не должен быть связан с файлом;
8. read fd смещение размер – прочитать данные из файла по заданному смещению указанного размера;
9. write fd смещение размер – записать данные в файл по заданному смещению указанного размера;
10. link имя1 имя2 – создать ссылку с именем имя2 на существующий файл с именем имя1;
11. unlink имя – уничтожить ссылку с заданным именем;
12. truncate имя размер – изменить размер файла, если размер файла увеличивается, то неинициализированные данные равны нулю.
13. mkdir имя – создать директорию с указанным именем;
14. rmdir имя – уничтожить пустую директорию с указанным именем (директория не должно содержать каких либо ссылок, за исключением предопределённых "." и "..");
15. cd имя – сменить текущую рабочую директорию;
16. pwd – вывести текущую рабочую директорию;
17. symlink имя1 имя2 – создать символическую ссылку с именем имя2 на путевое имя имя1.

**Листинг программы:**

**public class** FileSystem {  
  
 **public static** Scanner *scan* = **new** Scanner(System.***in***);  
 **private** Directory **rootDir**=**new** Directory();  
 **private** Directory **currDir**=**rootDir**;  
 **private final int MAX\_DESCRIPTORS**=100;  
 **private int descriptorsAmount**=0;  
 **private** HashMap<Integer,File> **opened**=**new** HashMap<>();  
 **public void** close(**int** id){  
 **opened**.remove(id);  
 }  
 **public void** open(String path){  
 Random r=**new** Random();  
 Node n=getNodeInPath(path);  
 **if**(n.getClass()==File.**class**){  
 File f=(File)n;  
 **int** fd=r.nextInt(1000);  
 **opened**.put(fd,f);  
 System.***out***.println(**"OPENED FILE WITH FD "**+fd);  
 }**else**{  
 System.***out***.println(**"ERROR! THIS IS NOT A FILE "**+n.getClass());  
 }  
 }  
 **public void** writefd(**int** offset,Block [] data){  
  
 }  
  
 **public void** cd(String path){  
 **currDir**=(Directory)getNodeInPath(path);  
 }  
 **public** String getName(String path){  
 String [] args=path.split(**"/+"**);  
 **return** args[args.**length**-1];  
 }  
 **public void** link(String fullname1,String fullname2){  
 Node n=getNodeInPath(fullname1);  
 String name=getName(fullname2);  
 Directory dir=getDir(fullname2);  
 **if**(n.getClass()==File.**class**){  
 File f=(File)n;  
 **if**(dir==**null**){  
 dir=**currDir**;  
 }  
 **if**(f.addLink(dir)){  
 dir.addNode(f,name);  
 }**else**{  
 System.***out***.println(**"TOO MANY LINKS TO THIS FILE"**);  
 }  
 }**else**{  
 System.***out***.println(**"IT IS IMPOSSIBLE TO ADD LINK TO THIS NODE"**);  
 }  
 }  
 **public void** symlink(String n1,String n2){  
 Directory dir=getDir(n2);  
 String name=getName(n2);  
 Symlink sl=**new** Symlink(n1);  
 **if**(dir==**null**){  
 **currDir**.addNode(sl, name);  
 }**else**{  
 dir.addNode(sl, name);  
 }  
 }  
 **public void** rmDir(String path){  
 Directory dir=(Directory)getNodeInPath(path);  
 **if**(dir.getAllNodes().size()>2){  
 System.***out***.println(**"IT IS IMPOSSIBLE TO REMOVE THIS DIR"**);  
 }**else**{  
 **boolean** is=**true**;  
 **for**(String key:dir.getAllNames()){  
 **if**(!key.equals(**"."**)&& !key.equals(**".."**)){  
 is=**false**;  
 }  
 }  
 **if**(is){  
 Directory parrent=(Directory)dir.getNode(**".."**);  
 **if**(dir!=**rootDir**){  
 parrent.remove(getName(path));  
 System.***out***.println(**"REMOVE DIR "**+getName(path));  
 }  
 **else**{  
 System.***out***.println(**"IT IS IMPOSSIBLE TO REMOVE THIS DIR"**);  
 }  
 }  
 **else**{  
 System.***out***.println(**"IT IS IMPOSSIBLE TO REMOVE THIS DIR"**);  
 }  
 }  
 }  
 **public void** ls(String path){  
 Directory node=(Directory)getNodeInPath(path);  
 **for**(Map.Entry<String, Node> entry:node.getAllNodes()){  
 **if**(entry.getValue().getClass()==File.**class**){  
 File f=(File)entry.getValue();  
 System.***out***.println(f.toString());  
 }  
 }  
 }  
 **public void** filestat(**int** id){  
 System.***out***.println(**opened**.get(id).toString());  
 }  
 **private** Directory getDir(String path){  
 String [] args=path.split(**"/+"**);  
 Directory dir=**null**;  
  
 **if**(args.**length**>1) {  
 **if** (args[0].equals(**".."**)) {  
 dir=(Directory)**currDir**.getNode(**".."**);  
 **for**(**int** i=1;i<args.**length**-1;i++) {  
 dir=(Directory)dir.getNode(args[i]);  
 }  
 }**else**{  
 **if**(args[0].equals(**"."**)){  
 dir=(Directory)**currDir**.getNode(**"."**);  
 **for**(**int** i=0;i<args.**length**-1;i++) {  
 dir=(Directory)dir.getNode(args[i]);  
 }  
  
 }**else**{  
 **if**(args[0].equals(**"~"**)){  
 dir=**rootDir**;  
 **for**(**int** i=0;i<args.**length**-1;i++) {  
 dir=(Directory)dir.getNode(args[i]);  
 }  
  
 }**else**{  
 dir=**currDir**;  
 **for**(**int** i=0;i<args.**length**-1;i++) {  
 dir=(Directory)dir.getNode(args[i]);  
 }  
 }  
  
 }  
  
 }  
 }  
 **return** dir;  
 }  
 **private** Node getNodeInPath(String path){  
 System.***out***.println(**"PATH "**+path);  
 String [] args=path.split(**"/+"**);  
 Node res=**null**;  
 Directory dir=getDir(path);  
 **if** (dir != **null**) {  
 res=dir.getNode(args[args.**length**-1]);  
  
 }**else**{  
 **if**(args[0].equals(**"~"**)){  
 dir=**rootDir**;  
 res=dir;  
 }**else**{  
 res=**currDir**.getNode(args[args.**length**-1]);  
 }  
 }  
 **if**(res.getClass()==Symlink.**class**){  
 System.***out***.println(**"TRANSITION TO SYMLINK"**);  
 Symlink link=(Symlink)res;  
 **if**(!link.isEmpty()){  
 String p=link.getPath();  
 System.***out***.println(**"PATH "**+p);  
 getNodeInPath(p);  
 }**else**{  
 System.***out***.println(**"SYMLINK LEADS TO NOWHRERE"**);  
 }  
  
 }  
 **return** res;  
 }  
 **public void** mkdir(String dirName){  
 String name=getName(dirName);  
 Directory parent=getDir(dirName);  
 Directory dir=**null**;  
 **if**(parent==**null**){  
 dir=**new** Directory(**currDir**);  
 **currDir**.addDir(dir, name);  
 }**else**{  
 dir=**new** Directory(parent);  
 parent.addDir(dir,name);  
 }  
 }  
 **public void** create(String fullname){  
 String name=getName(fullname);  
 Directory dir=getDir(fullname);  
 **if**(dir==**null**){  
 dir=**currDir**;  
 }  
 **if**(**descriptorsAmount**+1<=**MAX\_DESCRIPTORS**){  
 File f=**new** File(dir);  
 dir.addNode(f,name);  
 **descriptorsAmount**++;  
 }  
 }  
 **public void** pwd(){  
 **for**(Map.Entry<String, Node> entry:**currDir**.getAllNodes()){  
 System.***out***.println(entry.getKey());  
 }  
 System.***out***.println(**"AMOUNT OF NODES: "**+**currDir**.getAllNodes().size());  
 }  
 **public** ArrayList<Character> readfd(**int** fd,**int** offset,**int** size){  
 ArrayList<Character> res=**new** ArrayList<>();  
 File file=**opened**.get(fd);  
 ArrayList<Integer> links=file.getBlockLinks();  
 **int** globalOffset=offset/Block.***MAX\_SIZE***;  
 **int** localOffset=offset%Block.***MAX\_SIZE***;  
 **if**(localOffset!=0){  
 globalOffset++;  
 }  
 **int** endPosition=offset+size;  
 **int** endBlock=endPosition/Block.***MAX\_SIZE***;  
 **int** endOffset=endPosition%Block.***MAX\_SIZE***;  
 **if**(endOffset!=0){  
 endBlock++;  
 }  
 **int** counter=globalOffset;  
 **while**(counter<=endBlock){  
 **char** [] data=**null**;  
 **if**(counter==globalOffset){  
 data=Memory.*getInstance*().getBlock(file.getLink(counter)).getDataWithLeftOffset(localOffset);  
 }**else** {  
 **if**(counter==endBlock){  
 data=Memory.*getInstance*().getBlock(file.getLink(counter)).getDataWithRightOffset(endOffset);  
 }**else**{  
 data=Memory.*getInstance*().getBlock(file.getLink(counter)).getData();  
 }  
 }  
 **for**(**int** i=0;i<data.**length**;i++){  
 res.add(data[i]);  
 }  
 counter++;  
 }  
 **return** res;  
 }  
 **public void** writefd(**int** fd,**int** offset,**int** size,**char** [] data){  
 File file=**opened**.get(fd);  
 **int** globalOffset=offset/Block.***MAX\_SIZE***;  
 **int** localOffset=offset%Block.***MAX\_SIZE***;  
 **if**(localOffset!=0){  
 globalOffset++;  
 }  
 **int** endPosition=offset+size;  
 **int** endBlock=endPosition/Block.***MAX\_SIZE***;  
 **int** endOffset=endPosition%Block.***MAX\_SIZE***;  
 **if**(endOffset!=0){  
 endBlock++;  
 }  
 **int** counter=globalOffset;  
 file.expand(Memory.*getInstance*().getBlocks(endBlock));  
 **int** pos=0;  
 **while**(counter<=endBlock){  
 **int** len=Block.***MAX\_SIZE***;  
 **if**(counter==globalOffset){  
 len=Block.***MAX\_SIZE***-localOffset;  
 Memory.*getInstance*().getBlock(file.getLink(counter)).setData(Arrays.*copyOfRange*(data,pos,pos+len));  
 }**else** {  
 **if**(counter==endBlock){  
 len=Block.***MAX\_SIZE***-endOffset;  
 Memory.*getInstance*().getBlock(file.getLink(counter)).setData(Arrays.*copyOfRange*(data,pos,pos+len));  
 }**else**{  
 Memory.*getInstance*().getBlock(file.getLink(counter)).setData(Arrays.*copyOfRange*(data,pos,pos+len));  
 }  
 }  
 pos+=len;  
 counter++;  
  
 }  
 }  
 **public void** truncate(**int** fd,**int** size){  
 File file=**opened**.get(fd);  
 **int** blockAmount=size/Block.***MAX\_SIZE***;  
 **if**(file.getSize()<blockAmount){  
 ArrayList<Integer> links=Memory.*getInstance*().getBlocks(blockAmount-file.getSize());  
 file.expand(links);  
 }**else**{  
 file.truncate(file.getSize()-blockAmount);  
 }  
 }  
 **public void** mount(String fullPath){  
 java.io.File file=**new** java.io.File(fullPath);  
 **if**(file.exists()){  
 **try** {  
 FileInputStream fi=**new** FileInputStream(fullPath);  
 ObjectInputStream ois=**new** ObjectInputStream(fi);  
 **rootDir**=(Directory)ois.readObject();  
 } **catch** (Exception e ) {  
 e.printStackTrace();  
 }  
  
 }**else**{  
 System.***out***.println(**"THIS FILESYSTEM DOES'NT EXIST"**);  
 }  
 }  
 **public void** unlink(String fullName){  
 Node node=getNodeInPath(fullName);  
 String name=getName(fullName);  
 **if**(node.getClass()==File.**class**){  
 File fl=(File)node;  
 **for**(Node n:fl.getLinks()){  
 **if**(n.getClass()==Directory.**class**){  
 Directory dir=(Directory)n;  
 dir.remove(name);  
 }  
 }  
 }**else**{  
 System.***out***.println(**"THIS IS NOT A FILE"**);  
 }  
 }  
 **public void** unmount(){  
 **rootDir**=**null**;  
 }  
 **public void** save(String fullPath){  
 FileOutputStream fos = **null**;  
 **try** {  
 fos = **new** FileOutputStream(fullPath);  
 ObjectOutputStream oos = **new** ObjectOutputStream(fos);  
 oos.writeObject(**rootDir**);  
 oos.flush();  
 oos.close();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 **public static void** main(String ...args){  
  
 FileSystem fs=**new** FileSystem();  
  
 **while**(**true**){  
 String cmd = *scan*.nextLine() ;  
 String [] arguments=cmd.split(**"\\s+"**);  
 **switch** (arguments[0]){  
 **case "mount"**:  
 fs.mount(arguments[1]);  
 **break**;  
 **case "save"**:  
 fs.save(arguments[1]);  
 **break**;  
 **case "unmount"**:  
 fs.unmount();  
 **break**;  
 **case "mkdir"**:  
 fs.mkdir(arguments[1]);  
 **break**;  
 **case "pwd"**:  
 fs.pwd();  
 **break**;  
 **case "cd"**:  
 fs.cd(arguments[1]);  
 **break**;  
 **case "rmdir"**:  
 fs.rmDir(arguments[1]);  
 **break**;  
 **case "symlink"**:  
 fs.symlink(arguments[1], arguments[2]);  
 **break**;  
 **case "create"**:  
 fs.create(arguments[1]);  
 **break**;  
 **case "link"**:  
 fs.link(arguments[1], arguments[2]);  
 **break**;  
 **case "unlink"**:  
 fs.unlink(arguments[1]);  
 **break**;  
 **case "open"**:  
 fs.open(arguments[1]);  
 **break**;  
 **case "close"**:  
 fs.close(Integer.*parseInt*(arguments[1]));  
 **break**;  
 **case "ls"**:  
 fs.ls(arguments[1]);  
 **break**;  
 **case "filestat"**:  
 fs.filestat(Integer.*parseInt*(arguments[1]));  
 **break**;  
 **case "truncate"**:  
 fs.truncate(Integer.*parseInt*(arguments[1]),Integer.*parseInt*(arguments[2]));  
 **break**;  
 **case "writefd"**:  
 fs.writefd(Integer.*parseInt*(arguments[1]),Integer.*parseInt*(arguments[2]),Integer.*parseInt*(arguments[3]),**new char**[100]);  
 **break**;  
 **case "exit"**:  
 System.*exit*(0);  
 **break**;  
 }  
 }  
  
 }  
 }

**public class** Block **implements** Serializable {  
 **public static final int *MAX\_SIZE***=8;  
 **private char** [] **data**=**new char**[***MAX\_SIZE***];  
 **private boolean isBusy**;  
 **public boolean** isBusy(){  
 **return isBusy**;  
 }  
 **public void** setBusy(**boolean** isBusy){  
 **this**.**isBusy**=isBusy;  
 }  
 **public void** createData(){  
 **data**=**new char**[***MAX\_SIZE***];  
 }  
 **public void** setData(**char** [] data){  
 **this**.**data**=data;  
 }  
 **public void** setData(**char** [] data,**int** offset){  
 **this**.**data**=**new char**[***MAX\_SIZE***];  
 **int** counter=0;  
 **for**(**int** i=offset;i<***MAX\_SIZE***;i++){  
 **this**.**data**[i]=data[counter];  
 counter++;  
 }  
 }  
 **public void** fillWihOffset(**char** [] data,**int** offset){  
 **for**(**int** i=offset;i<=offset;i++){  
 **this**.**data**[i]=data[i];  
 }  
 }  
 **public char** [] getDataWithLeftOffset(**int** offset){  
 **return** Arrays.*copyOfRange*(**data**,offset,***MAX\_SIZE***);  
 }  
 **public char** [] getDataWithRightOffset(**int** offset){  
 **return** Arrays.*copyOfRange*(**data**,0,offset);  
 }  
 **public char**[] getData(){  
 **return data**;  
 }  
}

**public class** Directory **extends** Node **implements** Serializable{  
 **private** HashMap<String,Node> **nodes**=**new** HashMap<>();  
 **private int id**;  
 *//for foot directory* **public** Directory(){  
 **nodes**.put(**"."**,**this**);  
 Random r=**new** Random();  
 **id**=r.nextInt(1000);  
 }  
 **public** Directory(Directory parent){  
 **nodes**.put(**"."**,**this**);  
 **nodes**.put(**".."**,parent);  
 Random r=**new** Random();  
 **id**=r.nextInt(1000);  
 }  
 **public void** addDir(Directory dir,String name){  
 **nodes**.put(name,dir);  
 }  
 **public void** addNode(Node node,String name){  
 **nodes**.put(name,node);  
 }  
 **public** Node getNode(String key){  
 **return nodes**.get(key);  
 }  
 **public** Set<Map.Entry<String,Node>> getAllNodes(){  
 **return nodes**.entrySet();  
  
 }  
 **public** Set<String> getAllNames(){  
 **return nodes**.keySet();  
 }  
 **public void** remove(String name){  
 **nodes**.remove(name);  
 }  
  
}

**public class** File **extends** Node **implements** Serializable{  
 **public static final int *MAX\_LINKS***=5;  
 **private int id**;  
 **private** ArrayList<Node> **links**=**new** ArrayList<>();  
 **private** ArrayList<Integer> **blockLinks**=**new** ArrayList<>();  
 **public void** expand(ArrayList<Integer> links){  
 **blockLinks**.addAll(links);  
 }  
 **public void** truncate(**int** blockAmount){  
 **for**(**int** i=**blockLinks**.size()-1;i>=**blockLinks**.size()-blockAmount;i--){  
 **int** link=**blockLinks**.get(i);  
 Memory.*getInstance*().freeBlock(link);  
 }  
 }  
 **public** File(Directory source){  
 Random r=**new** Random();  
 **id**=r.nextInt(1000);  
 **links**.add(source);  
 }  
 **public** ArrayList<Node> getLinks(){  
 **return links**;  
 }  
 **public** ArrayList<Integer> getBlockLinks(){  
 **return blockLinks**;  
 }  
 **public boolean** addLink(Node node){  
 **if**(**links**.size()+1<=***MAX\_LINKS***){  
 **links**.add(node);  
 **return true**;  
 }  
 **else**{  
 System.***out***.println(**"IT IS IMPOSSIBLE TO ADD LINK TO THIS FILE"**);  
 **return false**;  
 }  
 }  
 **public** String toString(){  
 **return id**+**" "**+getClass();  
 }  
 **public int** getSize(){  
 **return blockLinks**.size();  
 }  
 **public int** getLink(**int** index){  
 **return blockLinks**.get(index);  
 } **public class** Memory {  
 **private static** Memory *instance*;  
 **private** Block [] **storage** = **new** Block[1000];  
 **private** Memory(){  
 **for**(**int** i=0;i<**storage**.**length**;i++){  
 **storage**[i]=**new** Block();  
 }  
 };  
 **public static** Memory getInstance(){  
 **if**(*instance*==**null**){  
 *instance*=**new** Memory();  
 }  
 **return** *instance*;  
 }  
 **public** ArrayList<Integer> getBlocks(**int** amount){  
 **int** counter=0;  
 ArrayList<Integer> res=**new** ArrayList<Integer>();  
 **for**(**int** i=0;i<**storage**.**length**;i++){  
 **if**(!**storage**[i].isBusy()){  
 **storage**[i].setBusy(**true**);  
 res.add(i);  
 }  
 counter++;  
 **if**(counter>=amount){  
 **break**;  
 }  
 }  
 **return** res;  
 }  
 **public** Block getBlock(**int** index){  
 **return storage**[index];  
 }  
 **public void** freeBlocks(ArrayList<Integer> blockLinks){  
 **for**(**int** i:blockLinks){  
 **storage**[i].setBusy(**false**);  
 }  
 }  
 **public void** freeBlock(**int** index){  
 **storage**[index].setBusy(**false**);  
 }  
}  
}

**public abstract class** Node **implements** Serializable{  
}

**public class** Symlink **extends** Node **implements** Serializable {  
 **private** String **path**;  
 **public** String getPath(){  
 **return path**;  
 }  
 **public** Symlink(String s){  
 **path**=s;  
 }  
 **public void** unlink(){  
 **path**=**""**;  
  
 }  
 **public boolean** isEmpty(){  
 **if**(**path**.equals(**""**)) **return true**;  
 **return false**;  
 }  
}