Name = Yahya Kemal Kuyumcu

Id = 220104004106

CSE241 Homework-2 OS Documentation

Manual

OPERANDS AND HOW TO USE THEM

rmdir [name of directory]

1. Only removes directory type files.
2. Takes the name of the directory user wanna remove as an argument.
3. Directory which name is giving for should be in the same directory with operand.
4. Can also take a path to the directory to remove as well ex : newdir/cac
5. The directory desired to be removed is should be reachable with cd commands from current directory if user inputs an path as an argument.

rm [name of file]

1. Only removes regular and linker type files.
2. Takes the name of the file user wanna remove as an argument.
3. File which name is giving for should be in the same directory with operand.
4. Can also take a path to the file to remove as well ex : newdir/cac.f
5. The file desired to be removed is should be reachable with cd commands from current directory if user inputs an path as an argument.

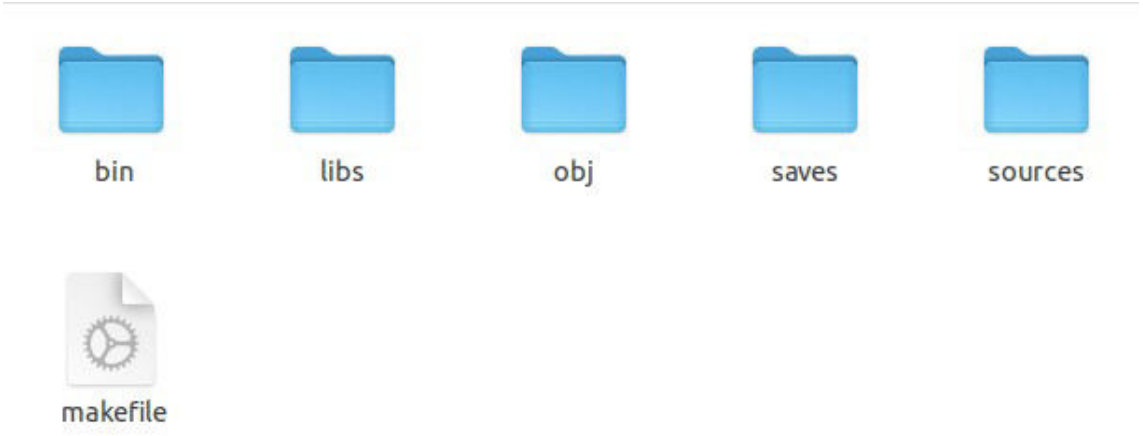
ls [-r or none]

1. If given none arguments ls shows the contents of the current directory with their name,created date time and size values.
2. If user gives -r as an argument ls command shows every content and every directory inside the current directory’s contents as well.

Cp [path to original file] [name of the copy]

1. If given path is pointing to an file or directory inside the fake os cp command takes an name argument but doesn’t use it so user should provide an name even if its an temporary value.
2. Copies the files to current directory where is cp called.
3. If user wants to point to an file inside fake os the path given should be reachable from current directory.
4. If user wants to copy a file from his original os, he should provide an path to that file, and a name value. Program can’t copy directory files from outside so this will only work with regular files.
5. If given a path inside the fake os, cp command can copy directory files too.

1) Program Structure



1)Bin Folder

Has the executable file.

2)Libs Folder

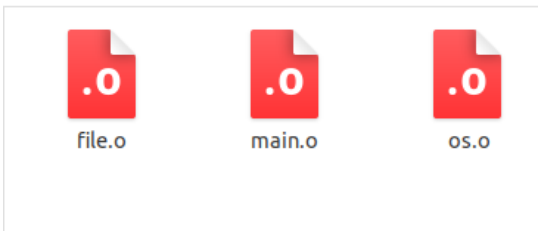
Has file.h and os.h header files.

File.h → Definitions of File, Directory,Regular File and LinkedFile classes.

Os.h → Definition of os class.

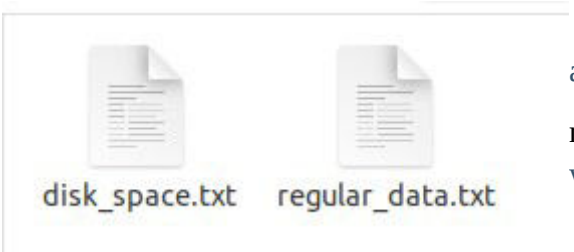
3)Obj Folder

Has .o files to link.



4)Saves Folder

Has disk_space.txt and regular_data.txt files for saving current state of os and datas before terminating the program.

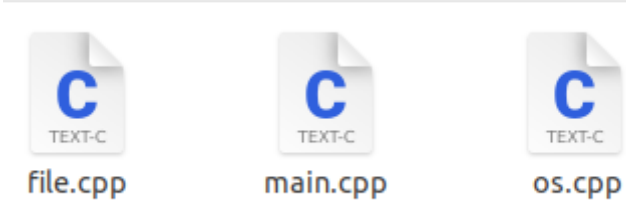


disk_space.txt → holds the directory's uniqiue data aswell ass size data and many other datas

regular_data.txt → holds the every regularFile's data which is copied from our os and inside this fake os

5)Sources Folder

Has file.cpp main.cpp os.cpp files.



2) FILE H

2.1) File Class

```
class File{//all implemented
    private:
    string name;
    size_t size;
    string created_dt;
    char type;
    shared_ptr<Directory> parent;
    int uniqiue_id;


    public:
        static void inc_id(){
            id_count = id_count +1 ;
        }

        File(string _n,char t,shared_ptr<Directory>parentF=nullptr,size_t size_send=1024):name(_n),size(size_send),type(t),parent(parentF){
            inc_id();
            uniqiue_id = File::id_count;
            time_t curr_time = time(nullptr);
            tm* timst = localtime(&curr_time);

            char buffer[80];
            strftime(buffer,sizeof(buffer),"%m-%d %H:%M",timst);
            created_dt = buffer;
        }
        //normal members
        //SETTERS
        void setName(const string& n){name = n;}
        void setType(const char c){type = c;}
        void setSize(size_t se){size = se;}
        void setUniq(const int u){uniqiue_id = u;}
        void setdt(const string& dt){created_dt = dt;}
        //GETTERS
        const int& getUnique()const{return uniqiue_id;}
        const char& getType()const{return type;}
        const size_t& getSize()const{return size;}
        const int getParentUniq()const;

        string getName()const{return name;}
        string getDT()const{return created_dt;}
        shared_ptr<Directory> getParent()const{return parent;}

        //virtuals
        virtual void cat()const = 0;
        virtual void ls()const = 0;
        virtual ~File(){}
        static int id_count;

};
```

Some member variables which every file in our filesystem should have, defined here in the private section of file class like name,size,created_dt(hold the date and time information of created file in string format) a type character(F,L,D) a shared ptr to parent directory and an unique id to make hierarchy for sub an parent when loading and saving from txt files.

File constructor takes an string, character,size value and a parent pointer to initialize.

Our static function and variable used for counting every created file to make them unique for hierarchy.

2.2) Directory Class

```
class Directory : public File{
    private:
        vector<shared_ptr<File>>contents;
    public:
        vector<shared_ptr< File> >& getContents(){
            return contents;
        }

        Directory(string n_,shared_ptr<Directory> parentF=nullptr,char t='D',size_t size=1024UL);

        void cat()const{
            this->ls();
        }
        void ls()const{
            for(auto x:contents){
                cout<<x->getType()<<" "<<x->getName()<<" "<<x->getDT()<<" size = "<<x->getSize() <<endl;
            }

        }
        void lsr(shared_ptr<Directory> dir,string prefix="")const{
            for(auto x: dir->getContents()){
                cout<<prefix<<x->getType()<<" "<<x->getName()<<" "<<x->getDT() << " size = "<<x->getSize()<<endl;
                if(dynamic_pointer_cast<Directory>(x)){
                    lsr(dynamic_pointer_cast<Directory>(x),prefix + " ");
                }
            }
        }
```

Directory class has member only one unique member variable different from its baseclass File which called contents, a vector of shared pointers that point to File objects. These pointers will be pointing to the child files of that directory.

The ls function prints out the features and names of the contents of that Directory.

The cat function simply calls the ls function.

The lsr function is an recursive function that prints out the current directory's every subdirectory's contents and every file's data to terminal.

The directory's iterator

```
//ITERATOR
class Iterator{
    private:
        int index;
        const Directory& dir;

    public:
        Iterator(const Directory& _dir,int _index=0):index(_index),dir(_dir){}
        Iterator& operator++(){
            index++;
            return *this;
        }
        Iterator operator++(int){
            Iterator it1(dir,index);
            index++;

            return it1;
        }
        Iterator& operator--(){
            if(index == 0){
                cout<<"Dont try to decrement an iterator when its pointing to the first element.";
                return *this;
            }else{
                index--;
                return *this;
            }
        }
        Iterator operator--(int){
            if(index == 0){
                cout<<"Dont try to decrement an iterator when its pointing to the first element.";
                return *this;
            }else{
                Iterator t1(dir,index);
                index--;
                return t1;
            }
        }

        bool operator!=(const Iterator& other){
            return index == other.index;
        }

        File& operator*(){
            return *dir.contents[index];
        }
};
```

The directory class's iterator is defined and implemented inside of the class definition.Its overloading ++postfix,++prefix,—postfix,--prefix,!= and dereference operator.Its private datas are an integer which holds the current index on the Directory's contents vector and a constant Directory reference.

The dereference operator than dereferences the pointer that hold in the vector in that index.

Other member functions of Directory Class

```
~Directory();

Iterator begin();
Iterator end();
Directory(const Directory& other);
size_t calculateSIZE();
void rm(string);

void add_data(shared_ptr<File > newf);
```

The begin function returns an iterator that points to the first element of directory's contents vector and the end function is returns an iterator that points to the one past the last element of that vector.

The class has an virtual destructor and a copy constructor.

The calculateSIZE function is calculating the sum of the size of every content in the directory then changes the size variable if there is an change in directorys contents.

Rm function is takes an name checks if it is exists in that directory than removes the file from filesystem.

The add_data functions is takes an file shared pointer and adds it to the contents of the directory.

2.2) RegularFile Class

```
class RegularFile:public File{
    private:
    vector<char> data;

    public:
    //ITERATOR
    class Iterator{
        private:
        int index;
        const RegularFile& file;
        public:
        Iterator(const RegularFile& fi,int _i=0):index(_i),file(fi){

        }
        Iterator& operator++(){
            index++;
            return *this;
        }
        Iterator operator++(int){
            Iterator it = Iterator(file,index);
            index++;
            return it;
        }
        Iterator& operator--(){
            if(index == 0){
                cout<<"Dont try to decrement an iterator when its pointing to the first element.";
                    return *this;
            }else{
                index--;
                return *this;
            }
        }
        Iterator operator--(int){
            if(index == 0){
                cout<<"Dont try to decrement an iterator when its pointing to the first element.";
                return *this;
            }else{
                Iterator tl(file,index);
                index--;
                return tl;
            }
        }

        bool operator!=(const Iterator& other){
        return index == other.index;
        }

        const char& operator*(){
            return file.data[index];
        }
    };
    //members;
    public:
    const vector<char>& getData(){return data;}
    void cp();
    RegularFile(string,shared_ptr<Directory > parentF=nullptr,size_t st=1);
    ~RegularFile();
    RegularFile(const RegularFile& other);
    void setData(char);
    Iterator begin();
    Iterator end();
    void cat()const{
        for(auto x:data){
            cout<<x;
        }

    }

    void ls()const{
        cout<<getName() <<" is not a directory."<<endl;
    }

};
```

The regular file class has an unique private variable which is an vector of characters named data. This variable holds the every byte of the file which is we copy from our os to the filesystem.

The iterator class is basically defined and implemented same as the directory class but this time the iterators are pointing to the every byte of the data, class holds.

The Members

The getData function is returns the constant reference of data vector.

Cp function is copies data from original file to the RegularFile.

SetData function pushes one character to the data vector.

Begin and end functions returns iterators that points to the beginning and one after the last element of the data vector.

Cat function is implemented in the class definition, it prints out the whole vector data.

2.3) LinkerFile Class

```
class LinkerFile:public File{
    private:
        int linked_file_unique;
        shared_ptr<File> linked_file;
    public:
        //constructor and destroctor
        ~LinkerFile();
        LinkerFile(string _n,shared_ptr<Directory >parentF,shared_ptr<File >lf,size_t size = 0);
        LinkerFile(const LinkerFile& other);
        //members
        void set_lk_unique(int iq){linked_file_unique = iq;}

        const int getLinkedUnique()const{return linked_file_unique;}
        void cat()const;
        void ls()const;

        void setLinked(shared_ptr<File>& linked){linked_file = linked;}
        const shared_ptr<File>& getLinked()const{return linked_file;}



};
```

LinkerFile class is inherited from base class File,it has 2 private members unique to LinkerFile class, linked_file_unique holds the unique integer of linked file to make loading from save files possible and linked_file pointer is pointing to linked file.

We also have some public getter's and setter's for our unique private members.

3)OS.H

```
struct FileInfo;
class OS{
    private:
        static const string operands[];

        const size_t disk_space;
        size_t used_space;
        string current_path;

        shared_ptr<Directory> root;
        shared_ptr< Directory> currentDir;

        void saveFileToFile(const shared_ptr<File>& file,ofstream& fileout,ofstream& outregular);
        void readit(vector<FileInfo>& fi);
        void loadBasicInfoFromFile(vector<FileInfo>& files, ifstream& in,ifstream& in_data);

        vector<string> tokenize_path(string);

        vector<string> tokenize_string(string);
        bool is_op_correct(string op);
        void operate(string input,vector<string>params);
        void change_Path();
    public:
        void saveToFile();
        void loadFromFile();

        static int parentdummy ;

        OS();
        void bash();
        bool loadOS();
        bool saveOS();
};
```

This os class is basically program's brain, every logic, input validation ,size calculation is done here.The constant disk_space variable holds 10mb size. The string variable current_path is basically only used for showing the user which folder is currently open.The root pointer doesn't change throughout the program, it points to the first directory of our filesystem but our currentDir pointer is changes everytime the user called cd operand.

FileInfo struct →

This struct is used for writing an reading from our save files.

,

```
struct FileInfo{
    string name;
    char type;
    size_t size;
    string date_time;
    int unique;
    int parentunique;
    int linkedinuqiue;
    size_t contentsSize;
    vector<char> data;

};
int OS::parentdummy = -1;
```

4) SOURCE FILES

4.1)File.cpp

1-DIRECTORY IMPLEMANTATIONS

```cpp
int File::id_count = 0;
const int File::getParentUniq()const{return parent->getUnique();}
//DIRECTORY FULLY IMPLEMENTED
Directory::Directory(string _n,shared_ptr<Directory> parentF,char t,size_t siz):File(_n,t,parentF,siz){

}

size_t Directory::calculateSIZE(){
    this->setSize(0);

    size_t samplesize = 0;

    for(auto x:this->contents){
        if(dynamic_pointer_cast<Directory>(x)){
            samplesize += (dynamic_pointer_cast<Directory>(x))->calculateSIZE();
        }else{
            samplesize += x->getSize();
        }
    }
    samplesize += 1024;

    // Update the size of the directory
    this->setSize(samplesize);

    return samplesize;
}

Directory::Iterator Directory::begin(){
    return Iterator(*this,0);
}
Directory::Iterator Directory::end(){
    return Iterator(*this,this->getContents().size());
}

void Directory::add_data(shared_ptr<File> newF){
    this->contents.push_back(newF);
    this->calculateSIZE();
}
void Directory::rm(string name){
    auto temp = contents.end();
    for (auto a = contents.begin(); a != contents.end(); ++a) {
        if ((*a)->getName() == name) {
            temp = a;
            break;
        }
    }

    if (temp != contents.end()) {
        contents.erase(temp);

        this->calculateSIZE();
    } else {
        cerr << "Cant find the file to remove!" << endl;
    }
}
Directory::Directory(const Directory& other):File(other.getName(),'D',other.getParent(),other.getSize()){
    for(auto a:other.contents){
        this->add_data(a);
    }
}
Directory::~Directory(){
    contents.erase(contents.begin(),contents.end());
}
```

2-REGULAR FILE IMPLEMENTATIONS

```
RegularFile::RegularFile(string _n,shared_ptr<Directory> parentF,size_t sizel):
File(_n,'F',parentF,sizel)
{

}
RegularFile::RegularFile(const RegularFile& ot):File(ot.getName(),'F',ot.getParent(),ot.getSize()){
    for(auto a:ot.data){
        this->data.push_back(a);
    }
}
RegularFile::~RegularFile(){

}

void RegularFile::setData(char data){
    this->data.push_back(data);
}

RegularFile::Iterator RegularFile::begin(){
    return  Iterator(*this,0);
}
RegularFile::Iterator RegularFile::end(){
    return Iterator(*this,data.size());
}
```

3-LINKER FILE IMPLEMENTATIONS

```
LinkerFile::LinkerFile(string _n,shared_ptr<Directory >parentF,shared_ptr<File> lf,size_t st):File(_n,'L',parentF,24){
    this->linked_file = lf;
    if(linked_file != nullptr){
        linked_file_unique = linked_file->getUnique();
    }else{
        linked_file_unique = -5;
    }

}

void LinkerFile::cat()const{
    cout<<"This file softly linked to file -> "<<linked_file->getName()<<endl;
    linked_file->cat();
}
void LinkerFile::ls()const{

}

LinkerFile::~LinkerFile(){

    linked_file.reset();
}

LinkerFile::LinkerFile(const LinkerFile& other):File(other.getName(),'L',nullptr,other.getSize()){
    linked_file_unique = other.linked_file_unique;
    linked_file = other.linked_file;
}
```

4.2)OS.CPP

4.2.1)Saving Functions

```
void OS::saveToFile(){

    const string filename = "./saves/disk_space.txt";
    const string rf_outdata = "./saves/regular_data.txt";

    ofstream file(filename,ios::out | ios::binary);
    ofstream regulardata(rf_outdata,ios::out | ios::binary);
    if(!regulardata.is_open() || !file.is_open()){
        cerr<<"Error opening save file."<<endl;
        throw invalid_argument("Cant open the regular_data.txt or disk_space.txt");
    }
    saveFileToFile(root,file,regulardata);


    file.close();
    regulardata.close();
}


void OS::saveFileToFile(const shared_ptr<File>& file, ofstream& out, ofstream& regular_data_out) {
    out << file->getName() << endl;
    out << file->getType() << endl;
    out << file->getSize() << endl;
    out << file->getDT() << endl;
    out << file->getUnique() << endl;

    if (file->getParent() != nullptr) {
        out << file->getParent()->getUnique() << endl;
    } else {
        out << OS::parentdummy << endl;
    }

    if (auto l_f = dynamic_pointer_cast<LinkerFile>(file)) {
        if (auto linkedFile = l_f->getLinked()) {
            if (linkedFile->getUnique() == file->getUnique()) {
                out << OS::parentdummy << endl;
            } else {
                out << linkedFile->getUnique() << endl;
                out << "0" << endl;
            }
        } else {
            cerr << "Error: LinkerFile has a null linked file." << endl;

        }
    } else {
        out << OS::parentdummy << endl;  // dummy linker -1
    }

    if (auto d_f = dynamic_pointer_cast<Directory>(file)) {
        out << d_f->getContents().size() << endl;
    } else if (auto r_f = dynamic_pointer_cast<RegularFile>(file)) {
        out << r_f->getData().size() << endl;
        for (const char& data : r_f->getData()) {
            regular_data_out << data;
        }
        regular_data_out << endl;  // Add a newline after writing data
    }

    if (auto d_f = dynamic_pointer_cast<Directory>(file)) {
        for (auto& x : d_f->getContents()) {
            saveFileToFile(x, out, regular_data_out);
        }
    }
}
```

1. Program uses 2 different txt files to store, hierarchy and unique values of file objects and regular file data.
2. We use dummy values for writing and reading consistency.

4.2.2)Loading Functions

Reading datas into FileInfo struct.

```
void OS::loadBasicInfoFromFile(vector<FileInfo>& files, ifstream& in, ifstream& in_data) {
    while (true) {
        FileInfo newfile;
        string saat;
        string tarih;
        if (!(in >> newfile.name >> newfile.type >> newfile.size >> tarih >> saat >> newfile.unique)) {

            break;
        }
        tarih = tarih + " " +saat;
        newfile.date_time = tarih;

        size_t size = 0;
        if (newfile.type == 'F') {
            newfile.data.resize(newfile.size);
            in_data.read(newfile.data.data(), newfile.size);
        }

        in >> newfile.parentunique >> newfile.linkedinuqiue >> newfile.contentsSize;

        files.push_back(newfile);
    }

    readit(files);
}



void OS::loadFromFile(){
    ifstream in("./saves/disk_space.txt",ios::in | ios::binary);
    ifstream in_data_rf("./saves/regular_data.txt",ios::in | ios::binary);


    if(!in.is_open() || !in_data_rf.is_open()){
        throw runtime_error("Cant open filesystem save files.");
    }

    root->getContents().clear();
    vector<FileInfo> fileifos;
    loadBasicInfoFromFile(fileifos,in,in_data_rf);

    in.close();
}
```

The readit function to create files based on FileInfo vector.

```
void OS::readit(vector<FileInfo>& fi){
    vector<shared_ptr<File>> createdOnes;

    for(auto start = fi.begin();start != fi.end();start++){
        if(start->parentunique == -1){
            root->setdt(start->date_time);
            root->setSize(start->size);
            createdOnes.push_back(root);
        }else{
            shared_ptr<Directory> p_ptr;
            shared_ptr<File> link_ptr;
            for(auto& x: createdOnes){
                if(x->getUnique() == start->parentunique){
                    p_ptr = dynamic_pointer_cast<Directory>(x);
                }else if(start->type == 'L'){
                    if(x->getUnique() == start->linkedinuqiue){
                        link_ptr = x;
                    }
                }
            }


            if(start->type == 'D'){

                auto newdir = make_shared<Directory>(start->name,p_ptr,'D',start->size);
                newdir->setdt(start->date_time);
                newdir->setUniq(start->unique);
                createdOnes.push_back(newdir);
            }else if(start->type == 'F'){
                auto newf = make_shared<RegularFile>(start->name,p_ptr,start->size);
                newf->setUniq(start->unique);
                newf->setdt(start->date_time);
                for(auto m : start->data){
                    newf->setData(m);
                }
                createdOnes.push_back(newf);
            }else{
                auto newlinker = make_shared<LinkerFile>(start->name,p_ptr,link_ptr,start->size);
                newlinker->setdt(start->date_time);
                newlinker->setUniq(start->unique);
                newlinker->set_lk_unique(start->linkedinuqiue);
                createdOnes.push_back(newlinker);
            }
        }
    }
    for(auto first= createdOnes.begin();first != createdOnes.end();first++){
        for(auto second = first + 1;second != createdOnes.end();second++ ){
            if((*first)->getUnique() == (*second)->getParentUniq()){
                dynamic_pointer_cast<Directory>((*first))->add_data(*second);
            }
            if(auto m = dynamic_pointer_cast<LinkerFile>(*second)){
                if((*first)->getUnique() == m->getLinkedUnique()){
                    m->setLinked(*first);
                }
            }

        }
    }

    root->calculateSIZE();
}
```

4.2.3) The Copying Algorithm

Inside the os

```
}else if(input == "cp"){

    bool is_inside_os = false;
    shared_ptr<Directory> d;
    if(params[0].find('/') != string::npos){
        vector<string>tokens = tokenize_path(params[0]);
        for(auto a: currentDir->getContents()){
            if(a->getName() == *tokens.begin() ){
                is_inside_os = true;
                break;
            }
        }
        if(is_inside_os){
            shared_ptr<Directory> original_dir = currentDir;
            string name_temp = *(tokens.end() - 1 );
            tokens.pop_back();
            operate("cd",tokens);
            for(auto x : currentDir->getContents()){
                if(x->getType() == 'D'){
                    if(x->getName() == name_temp){
                        if(x->getSize() + root->getSize() > 10*1024*1024){
                            cerr<<"Not enough disk space!"<<endl;
                            return ;
                        }
                    d = copyDirectory(*(dynamic_pointer_cast<Directory>(x)));
                    d->setParentPTR(original_dir);
                    original_dir->add_data(d);
                    break;
                    }
                }else if(x->getType() == 'F'){
                    if(x->getName() == name_temp){
                        if(x->getSize() + root->getSize() > 10*1024*1024){
                            cerr<<"Not enough disk space!"<<endl;
                            return ;
                        }
                    shared_ptr<RegularFile> rf = copyFile(*(dynamic_pointer_cast<RegularFile>(x)));
                    rf->setParentPTR(original_dir);
                    original_dir->add_data(rf);
                    break;
                    }
                }else if(x->getType() == 'L'){
                    if(x->getName() == name_temp ){
                        if(x->getSize() + root->getSize() > 10*1024*1024){
                            cerr<<"Not enough disk space!"<<endl;
                            return ;
                        }
                        shared_ptr<LinkerFile> lf = copyLinker(*(dynamic_pointer_cast<LinkerFile>(x)));
                        lf->setParentPTR(original_dir);
                        original_dir->add_data(lf);
                        break;
                    }
                }
            }
            currentDir = original_dir;
            change_Path();
            return;
        }
    }
```

COPYING ALGORITHMS

```
}

shared_ptr<LinkerFile> copyLinker(const LinkerFile& baseLinker){
    shared_ptr<LinkerFile> newL = make_shared<LinkerFile>(baseLinker.getName(),nullptr,baseLinker.getLinked(),24);
    return newL;
}
shared_ptr<RegularFile> copyFile(const RegularFile& baseFile){
    shared_ptr<RegularFile> newF = make_shared<RegularFile>(baseFile.getName(),nullptr,baseFile.getSize());
    for(auto a : baseFile.getData()){
        newF->setData(a);
    }
    return newF;
}

shared_ptr<Directory> copyDirectory(const Directory& baseDir) {
    shared_ptr<Directory> newDir = make_shared<Directory>(baseDir.getName() + "_cpy",nullptr,'D',baseDir.getSize());
    for (const auto& file : baseDir.getContents()) {
        if (file->getType() == 'F') {
            shared_ptr<RegularFile> newF = copyFile(*(dynamic_pointer_cast<RegularFile>(file)));
            newF->setParentPTR(newDir);
            newDir->add_data(newF);
        } else if (file->getType() == 'L') std::shared_ptr<LinkerFile> copyLinker(const LinkerFile &baseLinker)
            shared_ptr<LinkerFile> newL = copyLinker(*(dynamic_pointer_cast<LinkerFile>(file)));
            newL->setParentPTR(newDir);
            newDir->add_data(newL);
        } else if (file->getType() == 'D') {
            shared_ptr<Directory> newD = copyDirectory(*(dynamic_pointer_cast<Directory>(file)));
            newD->setParentPTR(newDir);
            newDir->add_data(newD);
        }
    }

    return newDir;
}
```

The copy directory function works recursively to create every subdir’s contents correctly.

Copying from Original OS

```
    if(!is_inside_os){
        for(auto x : currentDir->getContents()){
            if(x->getName() == params[1]){
                params[1] = params[1] + "_" + to_string(rand() % 150);
            }
        }

        ifstream input(params[0],ios::binary);
        if(!input.is_open()){
            throw runtime_error("Error oppening file.");
        }

        input.seekg(0,std::ios::end);
        size_t file_bytes = static_cast<size_t>(input.tellg());
        input.seekg(0,ios::beg);

        if(root->getSize() + file_bytes >= disk_space){
            cerr<<"Not enough disk space to copy  file -> "<<params[0]<<" remove some file."<<endl;
            cerr<<"Root size : "<<root->getSize()<<" disk space "<<disk_space<<" file_bytes "<<file_bytes<<endl;
            return;
        }

        shared_ptr<RegularFile> newRF = make_shared<RegularFile>(params[1],currentDir,file_bytes);
        char byte;

        while(input.get(byte)){
            newRF->setData(byte);
        }

        input.close();
        currentDir->add_data(newRF);
        currentDir->setSize(currentDir->calculateSIZE());
    }
```

Save data formatting

```
root
D
407613
01-07 01:37
1
-1
-1
7
file_cpy
D
48753
01-07 01:47
6
1
-1
2
new.jpeg
F
46672
01-07 01:47
7
6
-1
46672
main.k
F
1057
01-07 01:47
8
6
-1
1057
```

Each file in the system is represented by a sequence of eight elements. The first element is the file name, followed by a character indicating the file type: 'D' for directories, 'F' for RegularFiles, and 'L' for LinkerFiles. The third element denotes the file's size, while the fourth element captures the creation date and time. The fifth element holds the unique identifier of the file, and the sixth element stores the unique identifier of its parent directory. For the root directory, this parent identifier is set to -1.

The seventh element corresponds to the unique identifier of a linked file, applicable to LinkerFiles; for other file types, it is set to -1 as a dummy value. The eighth and final element represents the contents size for directories. For RegularFiles, it equals the size of the data vector, and for LinkerFiles, it is set to zero. This structured arrangement facilitates the systematic reading and interpretation of file information during the loading process.

OPERANDS AND HOW TO USE THEM

rmdir [name of directory]

6. Only removes directory type files.
7. Takes the name of the directory user wanna remove as an argument.
8. Directory which name is giving for should be in the same directory with operand.
9. Can also take a path to the directory to remove as well ex : newdir/cac
10. The directory desired to be removed is should be reachable with cd commands from current directory if user inputs an path as an argument.

rm [name of file]

1. Only removes regular and linker type files.
2. Takes the name of the file user wanna remove as an argument.
3. File which name is giving for should be in the same directory with operand.
4. Can also take a path to the file to remove as well ex : newdir/cac.f
5. The file desired to be removed is should be reachable with cd commands from current directory if user inputs an path as an argument.

ls [-r or none]

1. If given none arguments ls shows the contents of the current directory with their name,created date time and size values.
2. If user gives -r as an argument ls command shows every content and every directory inside the current directory’s contents as well.

Cp [path to original file] [name of the copy]

1. If given path is pointing to an file or directory inside the fake os cp command takes an name argument but doesn’t use it so user should provide an name even if its an temporary value.
2. Copies the files to current directory where is cp called.
3. If user wants to point to an file inside fake os the path given should be reachable from current directory.
4. If user wants to copy a file from his original os, he should provide an path to that file, and a name value. Program can’t copy directory files from outside so this will only work with regular files.
5. If given a path inside the fake os, cp command can copy directory files too.

Link [path to the desired file or directory] [linkfile name]

1. The path is have to be reachable from current directory.
2. User can create link files backwards or forward in hierarchy.
3. Have to provide linkfile name

Cd [path to desired directory]

1. The provided path should be reachable from current directory.
2. Can use “..” or “.” parameters too.
3. Updates currentDir variable in os class.

Cat [path to file or directory]

1. The path is have to be reachable from current directory.
2. Can be used on directory files as well, it simply calls ls command for them.
3. When used on regular files it prints out the data stored in regular file.
4. When used on linker files it calls the cat command on linked file.

Mkdir [name of the directory which will be created]

1. In the current directory program will create an directory by given name.
2. Directories can only be created inside the current directory.
3. Command doesn’t take any path.

Exit

1. Closes the program.
2. Before closing the program saves all the data.