Background

## File Systems

A file system is a data structure that an operating system uses to control how data is stored and retrieved. A file system consists mostly of directories (folders) and regular files, and users use file systems to organise their data by creating, deleting, moving and renaming files.

Due to their hierarchical nature, most file systems can be viewed as trees where each node represents a file. Internal nodes (nodes that have at least one child) correspond to directories, while leaves correspond to either regular files or empty directories. Here is an example of a file system:

## ... ... ...

/(root)

bin

etc

home

init

tmp

ls

mv passwd ssh

jas

tmp.123 tmp.456

tmpdir

* . - this refers to the tmp directory.
* .. - this refers to the parent directory of the tmp directory, i.e. the root directory.
* tmpdir - this refers to the tmpdir file which is in the tmp directory.
* ../bin/ls - this refers to the ls file in the bin directory.
* ../home/jas - this refers to the jas file in the home directory.
* ../../../home/jas - since the parent directory of the root directory is itself, this still refers to the jas file in the home directory.

Note that due to the existence of the files . and .. in every directory and the fact that paths can be absolute or relative, it is possible for many different paths to refer to the same file. For example, in the above file system, the following paths are equivalent:

* /etc/passwd
* ../etc/passwd
* ../etc/./passwd
* ./../../bin/../etc/ssh/../passwd

Therefore, we introduce the notion of a canonical path. The canonical path of a file is the unique absolute path of the file that does not contain . or ... For example, the canonical path of the passwd file in the above image is /etc/passwd.

###### Path Prefixes

Here we will describe what is meant by a prefix or proper prefix of a path, which are terms used later in the specification.

A path PP is a prefix of a path QQ if PP can be extended by zero or more filenames to obtain Q. For example, the prefixes of the path /home/jas/cs/lecs are: /, /home, /home/jas, /home/jas/cs and /home/jas/cs/lecs. The prefixes of the path ../cs/labs are: .., ../cs and ../cs/labs.

A path PP is a proper prefix of a path QQ if PP can be extended by one or more filenames to obtain Q. For example, the proper prefixes of the path /home/jas/cs/lecs are: /, /home, /home/jas and /home/jas/cs.

In this task, you'll be implementing an ADT to simulate a simplfied Linux file system. Importantly, the file system will be in- memory, which means no real files are actually created on disk. Instead, you will use structs to simulate files, so these so-called "files" will only exist in RAM and will no longer exist once the program terminates.

## Files

In Linux, there are many different types of files: regular files (such as text files, object files and executables), directories (that's right

- directories are files), symbolic links, sockets, named pipes, character devices and block devices. In this assignment, you won't need to deal with all of these types of files - you'll only need to handle regular files (specifically, text files) and directories.

## Home Directories

File systems usually contain a home directory for each user under the /home directory. For example, /home/jas would be the home directory for the user "jas". In Linux, if you use the cd command without any arguments, you'll be taken to your home directory. To simplify things in this assignment, you won't need to deal with home directories.

## Filenames

In Linux, filenames are case sensitive and may contain any character except forward slash (/), which is reserved as the separator between files and directories in a path. To simplify things in this assignment, you can assume that filenames contain only alphanumeric characters, dot (.), hyphen (-) and underscore (\_). In this assignment, you may make no assumptions about filename length - filenames can be arbitrarily long.

All directories implicitly contain two files named . (dot) and .. (dot dot). . refers back to the directory itself, while .. refers to the parent directory. These files are hidden, so they will not be listed if you use the ls command without the -a option. In this assignment, most commands (including ls) will not have options.

In Linux, filenames that begin with dot (.) are automatically hidden. In this assignment, you can assume that filenames won't start with ., except for . and .. which are implicitly contained in every directory. This means that you *could* implement the hidden file feature if you want, but it will not be tested.

## Paths

How do we describe the location of files and navigate around the file system? We use paths! A path is a sequence of filenames separated by forward slashes (/). Consecutive forward slashes are treated as if they were a single forward slash. There are two kinds of paths: absolute paths, which begin with a slash and describe the location of a file relative to the root directory (the directory at the top of the file system), and relative paths, which describe a location relative to the current working directory. The current working directory (cwd) is the directory that the program/terminal is currently working from, and can be changed with the cd command.

Here are some examples of absolute paths:

/ - this path refers the root directory.

/tmp - this path refers to the tmp file under the root directory.

/tmp/tmp.123 - this path refers to the tmp.123 file which is in the tmp directory, which is in the root directory.

/tmp//tmp.123 - this also refers to the tmp.123 file in the tmp directory, since consecutive forward slashes are treated as if they were a single forward slash.

Now suppose that the tmp directory in the image below is the current working directory. Here are some examples of relative paths:

. - this refers to the tmp directory.

## ...

/(root)

bin

etc

home

init

tmp

ls

mv passwd ssh

jas

tmp.123 tmp.456

tmpdir

... ...

.. - this refers to the parent directory of the tmp directory, i.e. the root directory.

tmpdir - this refers to the tmpdir file which is in the tmp directory.

../bin/ls - this refers to the ls file in the bin directory.

../home/jas - this refers to the jas file in the home directory.

../../../home/jas - since the parent directory of the root directory is itself, this still refers to the jas file in the home

directory.

Note that due to the existence of the files . and .. in every directory and the fact that paths can be absolute or relative, it is possible for many different paths to refer to the same file. For example, in the above file system, the following paths are equivalent:

### /etc/passwd

../etc/passwd

../etc/./passwd

./../../bin/../etc/ssh/../passwd

Therefore, we introduce the notion of a canonical path. The canonical path of a file is the unique absolute path of the file that does not contain . or ... For example, the canonical path of the passwd file in the above image is /etc/passwd.

Path Prefixes

Here we will describe what is meant by a *prefix* or *proper prefix* of a path, which are terms used later in the specification.

A path *P* is a prefix of a path *Q* if *P* can be extended by zero or more filenames to obtain *Q*. For example, the prefixes of the path

/home/jas/cs/lecs are: /, /home, /home/jas, /home/jas/cs and /home/jas/cs/lecs. The prefixes of the path ../cs/labs are: .., ../cs and ../cs/labs.

A path *P* is a proper prefix of a path *Q* if *P* can be extended by one or more filenames to obtain *Q*. For example, the proper prefixes of the path /home/jas/cs/lecs are: /, /home, /home/jas and /home/jas/cs.

Setting Up

**Makefile** a set of dependencies used to control compilation

**Fs.h** the interface to the File System ADT

**Fs.c** the implementation of the File System ADT (incomplete) **utility.h** the interface to your utility data structures and functions **utility.c** the implementation of your utility functions

**FileType.h** contains the definition of the FileType enum

**listFile.c** contains the listFile function, which prints filenames

**testFs.c** a main program to test the File System ADT

First, compile the original version of the files using the make command. You'll see that it produces two executables: testFs and testFsColored. If you use the provided listFile function to print out filenames (which we recommend), testFsColored will apply coloring to directory names, which will help you distinguish regular files and directories when running your tests in the terminal. testFs will print uncolored filenames.

Most of your file system implementation will go in Fs.c. Here you may define your own constants, structs and helper functions. We have also provided the files utility.c and utility.h where you can place data structures and functions that you think should go in a separate file. The only files you are allowed to submit are Fs.c, utility.c and utility.h, so all your code must be in these three files.

testFs.c is the main program that you will be using to test your implementation. At the moment it contains only one basic test, but you should add more tests to it. There are many example tests below that you could use.

Stage 0

# FsNew

The FsNew function has the signature:

Fs FsNew(void);

This function should allocate and initialise a new struct FsRep, create the root directory for the file system and make the root directory the current working directory. It should then return a pointer to the allocated struct FsRep.

# FsGetCwd

The FsGetCwd function has the signature:

void FsGetCwd(Fs fs, char cwd[PATH\_MAX + 1]);

This function should store the canonical path of the current working directory in the given cwd array. It can assume that the canonical path of the current working directory is no longer than PATH\_MAX characters.

# FsFree

The FsFree function has the signature:

void FsFree(Fs fs);

This function should free all memory associated with the given Fs. You may need to update this function as you work on each stage to free any new data structures that you created.

Stage 1

# FsMkdir

The FsMkdir function has the signature:

void FsMkdir(Fs fs, char \*path);

The function takes a path and creates a new directory at that path in the given file system. FsMkdir performs roughly the same function as the mkdir command in Linux.

## Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error message for the first error from the table that applies.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Description | Error Message | | | | | | | |
| A file already exists at the given path | mkdir: | cannot | create | directory | '*path*': | File |  | exists |
| A prefix of the path is a regular file | mkdir: | cannot | create | directory | '*path*': | Not | a | directory |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A proper prefix of the path does not exist | mkdir: | cannot | create | directory | '*path*': | No | such | file | or | directory |

Note that when you print an error message, you should replace *path* with the given path. For example, if the given path was

labs/lab01 and a file called labs/lab01 already exists, then the error message should be:

All error messages (including error messages in the rest of the functions) should be printed to stdout, which means you

should use printf to print them. Also note that when one of these errors occurs, the program should not exit - the function should simply return with the file system left unchanged.

mkdir: cannot create directory 'labs/lab01': File exists

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 Example 5 |
| Program:  int main(void) {  Fs fs = FsNew();  *// this is equivalent to FsMkdir(fs, "tmp"); because initially,*  *// the current working directory is the root directory*  FsMkdir(fs, "/tmp");  FsMkdir(fs, "/tmp/tmp.123"); FsMkdir(fs, "/usr");  FsMkdir(fs, "/bin");  *// see the section for FsTree for details*  FsTree(fs, NULL);  }  Expected output:  /  bin tmp  tmp.123  usr | |

FsMkfile

The FsMkfile function has the signature:

void FsMkfile(Fs fs, char \*path);

The function takes a path and creates a new empty regular file at that path in the given file system. This function does not have a direct equivalent command in Linux, but the closest command is touch, which can be used to create empty regular files, but also has other uses such as updating timestamps.

## Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error message for the first error from the table that applies.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Description | Error Message | | | | | | |
| A file already exists at the given path | mkfile: | cannot | create | file | '*path*': | File exists |  |
| A prefix of the path is a regular file | mkfile: | cannot | create | file | '*path*': | Not a directory |  |
| A proper prefix of the path does not exist | mkfile: | cannot | create | file | '*path*': | No such file or | directory |

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 Example 5 |
| Program:  int main(void) {  Fs fs = FsNew();  *// this is equivalent to FsMkdir(fs, "hello.c"); because initially,*  *// the current working directory is the root directory*  FsMkfile(fs, "/hello.c");  FsMkfile(fs, "world.c");  FsMkdir(fs, "/bin"); FsMkfile(fs, "bin/mkdir"); FsMkfile(fs, "bin/mkfile");  *// see the section for FsTree for details*  FsTree(fs, NULL);  }  Expected output:  /  bin  mkdir mkfile  hello.c world.c | |

FsCd

The FsCd function has the signature:

void FsCd(Fs fs, char \*path);

The function takes a path which may be NULL.

If the path is not NULL, the function should change the current working directory to that path.

If the path is NULL, it defaults to the root directory (not the home directory, since we do not have home directories in this assignment).

This function roughly corresponds to the cd command in Linux.

## Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error message for the first error from the table that applies.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Description | Error Message | | | | |
| A prefix of the path is a regular file | cd: | '*path*': | Not | a directory |  |
| A prefix of the path does not exist | cd: | '*path*': | No | such file or | directory |

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 Example 5 |
| Program: | |

int main(void) {

Fs fs = FsNew(); FsMkdir(fs, "/home"); FsCd(fs, "home");

FsMkdir(fs, "jas");

FsCd(fs, "jas"); FsMkdir(fs, "cs"); FsCd(fs, "cs"); FsMkdir(fs, "lectures"); FsMkdir(fs, "tutes"); FsMkdir(fs, "labs"); FsTree(fs, NULL);

}

Expected output:

/

home

jas

cs

labs lectures tutes

FsLs

The FsLs function has the signature:

void FsLs(Fs fs, char \*path);

The function takes a path which may be NULL.

If the path is not NULL and refers to a directory, then the function should print the names of all the files in that directory (except for . and ..) in ASCII order, one per line. If the path refers to a file, then the function should just print the given path if it exists in the file system.

If the path is NULL, it defaults to the current working directory. This function roughly corresponds to the ls command in Linux. Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error

message for the first error from the table that applies.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Description | Error Message | | | | | | |
| A *proper* prefix of the path is a regular file | ls: | cannot | access | '*path*': | Not | a directory |  |
| A prefix of the path does not exist | ls: | cannot | access | '*path*': | No | such file or | directory |

## Usage Examples

Example 1

Example 2 Example 3 Example 4 Example 5

Program:

int main() {

Fs fs = FsNew();

printf("---**\n**"); *// marker to separate output*

FsLs(fs, "/");

printf("---**\n**"); FsMkfile(fs, "hello.txt"); FsMkdir(fs, "tmp");

FsLs(fs, "/");

}

Expected output:

---

---

hello.txt tmp

FsPwd

The FsPwd function has the signature:

void FsPwd(Fs fs);

The function prints the canonical path of the current working directory. This function roughly corresponds to the pwd command in Linux.

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 |
| Program:  int main() {  Fs fs = FsNew(); FsPwd(fs); FsMkdir(fs, "home");  FsCd(fs, "home"); FsPwd(fs); FsMkdir(fs, "jas");  FsCd(fs, "jas"); FsPwd(fs);  }  Expected output:  /  /home  /home/jas | |

FsTree

The FsTree function has the signature:

void FsTree(Fs fs, char \*path);

The function takes one path which may be NULL. If the path is NULL, it defaults to the root directory.

The function prints the directory hierarchy of the given path in a structured way (see below). This function roughly corresponds to the tree command in Linux.

## Output Format

The first line of the output should contain the given path if it exists and refers to a directory. The following lines should display all the files under the given directory in ASCII order, one per line, with indentation to show which files are contained under which directories. Indentation should increase by 4 spaces per level. See the Usage Examples section for examples.

## Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error message for the first error from the table that applies.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Description | Error Message | | | | |
| A prefix of the path is a regular file | tree: | '*path*': | Not | a directory |  |
| A prefix of the path does not exist | tree: | '*path*': | No | such file or | directory |

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 |
| Program:  int main() {  Fs fs = FsNew(); FsMkfile(fs, "hello.txt"); FsMkfile(fs, "world.txt"); FsMkdir(fs, "bin"); FsMkfile(fs, "bin/ls"); FsMkfile(fs, "bin/pwd"); FsMkdir(fs, "home"); FsMkdir(fs, "home/jas");  FsMkfile(fs, "home/jas/todo.txt"); FsMkfile(fs, "home/jas/mail.txt"); FsTree(fs, "/home/jas");  printf("---**\n**"); *// marker to separate output*  FsTree(fs, NULL);  }  Expected output:  /home/jas  mail.txt todo.txt  ---  /  bin  ls pwd  hello.txt home  jas  mail.txt todo.txt  world.txt | |

Stage 2

FsPut

The FsPut function has the signature:

void FsPut(Fs fs, char \*path, char \*content);

The function takes a path and a string, and sets the content of the regular file at that path to the given string. If the file already had some content, then it will be overwritten.

## Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error message for the first error from the table that applies.

|  |  |
| --- | --- |
| Description | Error Message |
| The path refers to a directory | put: '*path*': Is a directory |
|  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A proper prefix of the path is a regular file | put: | '*path*': | Not | a directory | | | |
| A prefix of the path does not exist | put: | '*path*': | No | such | file | or | directory |

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 |
| Program:  int main() {  Fs fs = FsNew(); FsMkfile(fs, "hello.txt");  FsPut(fs, "hello.txt", "hello**\n**");  FsPut(fs, "./hello.txt", "world**\n**"); *// overwrites existing content*  }  This program has no expected output. | |

FsCat

The FsCat function has the signature:

void FsCat(Fs fs, char \*path);

The function takes a path and prints the content of the regular file at that path. This function roughly corresponds to the cat

command in Linux.

## Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error message for the first error from the table that applies.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Description | Error Message | | | | |
| The path refers to a directory | cat: | '*path*': | Is | a directory |  |
| A proper prefix of the path is a regular file | cat: | '*path*': | Not | a directory |  |
| A prefix of the path does not exist | cat: | '*path*': | No | such file or | directory |

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 |
| Program:  int main() {  Fs fs = FsNew(); FsMkfile(fs, "hello.txt");  FsPut(fs, "hello.txt", "hello**\n**");  FsCat(fs, "hello.txt");  FsPut(fs, "./hello.txt", "world**\n**"); *// overwrites existing content*  FsCat(fs, "/hello.txt");  }  Expected output:  hello world | |

FsDldir

The FsDldir function has the signature:

void FsDldir(Fs fs, char \*path);

The function takes a path which is expected to refer to a directory and deletes the directory if and only if it is empty. This function roughly corresponds to the rmdir command in Linux.

For simplicity, you may assume that the given path does not contain the current working directory. Note that this means that the given path will never be the root directory. You can handle this situation if you want (for completeness' sake), but it will not be tested.

## Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error message for the first error from the table that applies.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Description | Error Message | | | | | |
| The path refers to a non-empty directory | dldir: | failed | to | remove | '*path*': | Directory not empty |
| A prefix of the path is a regular file | dldir: | failed | to | remove | '*path*': | Not a directory |
| A prefix of the path does not exist | dldir: | failed | to | remove | '*path*': | No such file or directory |

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 |
| Program:  int main() {  Fs fs = FsNew(); FsMkdir(fs, "hello"); FsMkdir(fs, "hello/world"); FsTree(fs, NULL);  printf("---**\n**"); *// marker to separate output*  FsDldir(fs, "hello/world"); FsDldir(fs, "hello"); FsTree(fs, NULL);  }  Expected output:  /  hello  world  ---  / | |

FsDl

The FsDl function has the signature:

void FsDl(Fs fs, bool recursive, char \*path);

The function takes a path and deletes the file at that path. By default, the function refuses to delete directories; it will only delete a directory (and all of its contents recursively) if recursive is true. If the path refers to a regular file, then the recursive argument is irrelevant. This function roughly corresponds to the rm command in Linux, and recursive being true corresponds to the -r option being used in the rm command.

For simplicity, you may assume that the given path does not contain the current working directory. Note that this means that the given path will never be the root directory. You can handle this situation if you want (for completeness' sake), but it will not be tested.

Be careful when experimenting with the rm command in Linux - to avoid accidentally deleting important files (or your entire file system), always experiment in a temporary directory and never begin the path with slash. You can create a temporary directory by using the mktemp -d command. This command will create a temporary directory under the /tmp directory and then output its path, which you can then cd to.

## Errors

You must handle the following errors and produce error messages exactly as shown here. If multiple errors apply, only print an error message for the first error from the table that applies.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Description | Error Message | | | | | | |
| The path refers to a directory but recursive is false | dl: | cannot | remove | '*path*': | Is | a directory |  |
| A proper prefix of the path is a regular file | dl: | cannot | remove | '*path*': | Not | a directory |  |
| A prefix of the path does not exist | dl: | cannot | remove | '*path*': | No | such file or | directory |

## Usage Examples

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 |
| Program:  int main() {  Fs fs = FsNew(); FsMkdir(fs, "hello");  FsMkfile(fs, "hello/world.txt"); FsMkfile(fs, "abc.txt"); FsTree(fs, NULL);  printf("---**\n**"); *// marker to separate output*  FsDl(fs, true, "abc.txt"); FsTree(fs, NULL);  printf("---**\n**"); *// marker to separate output*  FsDl(fs, true, "hello"); FsTree(fs, NULL);  }  Expected output:  /  abc.txt hello  world.txt  ---  /  hello  world.txt  ---  / | |

Stage 3

Stage 3 implements the ability to copy files.

FsCp

The FsCp function has the signature:

void FsCp(Fs fs, bool recursive, char \*src[], char \*dest);

The function takes a NULL-terminated array of paths src and a path dest. If the src array contains exactly one path, then it should copy the file at src to dest. If the src array contains more than one path, then dest is expected to refer to a directory, and the function should copy all the files at the paths in the src array to the directory at dest. By default, the function does not copy directories - it should only copy directories if recursive is true.

This function roughly corresponds to the cp command in Linux.

## Errors

If you experiment with the cp command in Linux, you will find that many different errors are possible (more than 10). Since we don't want the focus of the assignment to be on handling errors, you can assume that the arguments given to FsCp will not result in an error. However, we've provided a list of possible error messages for your curiosity:

Possible errors for cp

## Usage Examples

We've provided examples where the src array contains exactly one path.

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 Example 5 |
| Program:  int main() {  Fs fs = FsNew(); FsMkfile(fs, "hello.txt");  FsPut(fs, "hello.txt", "hello**\n**");  FsMkfile(fs, "world.txt"); FsPut(fs, "world.txt", "world**\n**"); FsCat(fs, "world.txt");  printf("---**\n**");  char \*src[] = { "hello.txt", NULL }; FsCp(fs, false, src, "world.txt"); FsCat(fs, "world.txt");  printf("---**\n**");  FsTree(fs, NULL);  }  Expected output:  world  ---  hello  ---  /  hello.txt world.txt  Description: If the src and dest paths both refer to regular files, then the contents of the src file should simply be copied to the dest file, overwriting its contents. | |

We leave you to determine the specific behaviour when the src array contains more than one path. You can do this by experimenting with the cp command in Linux - the behaviour of your file system should mimic what Linux does.

When experimenting with the cp command, you should always create a temporary directory to experiment in to avoid accidentally overwriting your own files or exceeding your disk quota (if you are working on CSE).

Stage 4

Stage 4 implements the ability to move and rename files (in Linux, these are the same thing).

# FsMv

The FsMv function has the signature:

void FsMv(Fs fs, char \*src[], char \*dest);

The function takes a NULL-terminated array of paths src and a path dest. It should move all the files referred to by paths in src to

### dest.

This function roughly corresponds to the mv command in Linux.

For example, consider the file system shown in the Background section. If the bin directory is moved to the tmp directory, then the file system would now look like:

/(root)

etc

home

init

tmp

passwd ssh

jas

bin

tmp.123 tmp.456

tmpdir

...

...

...

ls

mv

This move would be achieved by the following piece of code in testFs.c:

char \*src[] = { "/bin", NULL };

FsMv(fs, src, "/tmp");

Errors

Like in Stage 3, to keep the focus on the implementation, you can assume that the arguments given to FsMv will not result in an error. However, for your curiosity, here are some of the possible errors that you could get from the Linux mv command:

Possible errors for mv

Like in Stage 3, we've provided examples where the src array contains exactly one path. It is up to you to determine the correct behaviour when the src array contains multiple paths.

|  |  |
| --- | --- |
| Example 1 | Example 2 Example 3 Example 4 Example 5 |
| Program:  int main() {  Fs fs = FsNew(); FsMkfile(fs, "hello.txt");  FsPut(fs, "hello.txt", "hello**\n**");  FsTree(fs, NULL); printf("---**\n**");  char \*src[] = { "hello.txt", NULL }; FsMv(fs, src, "world.txt"); FsTree(fs, NULL);  printf("---**\n**");  FsCat(fs, "world.txt");  }  Expected output: | |

Description: If the dest path does not exist, but all of its proper prefixes do exist, then the src file should simply be moved to the dest path.

/

hello.txt

---

/

world.txt

---

hello

---