UEFI File System Explorer

Mohammadreza Ahmaditeshnizi (98170646)
Matin Shoja (98101809)
Ali Abbasi (98105879)

February 1, 2024

Contents

Introduction

In this project, we were responsible for developing a UEFI program functioning as a file system explorer. Our primary goal was to design a user interface that allows for easy navigation and interaction with files and directories in the UEFI environment. This project involved a detailed study of UEFI specifications and understanding the principles of developing related applications. Utilizing the protocols available in the UEFI file system, we integrated files and directories. Additionally, developing the user interface, adding advanced features like file searching, filtering, and sorting, and developing error management mechanisms to handle potential exceptions and errors were part of the challenges of this project. Finally, we conducted extensive testing and validation of the program in various scenarios.

1 Prerequisites

Before starting, ensure your Linux distribution has the following packages installed:

- **NASM (Netwide Assembler)**: An assembler and disassembler for the Intel x86 architecture.
- **IASL (Intel ACPI Compiler/Decompiler)**: Used for Advanced Configuration and Power Interface (ACPI).
- **UUID-Dev (Universally Unique ID Library)**: Essential for generating unique identifiers in UEFI.
- **Python 3**: Necessary for running build infrastructure scripts.

1.1 Installing Packages on Ubuntu

Run the following commands in your terminal:

```
sudo apt-get install -y nasm iasl uuid-dev python3
sudo apt-get install -y python3-distutils
```

1.2 Cloning and Preparing EDK2 Repository

```
git clone https://github.com/tianocore/edk2
cd edk2
git submodule update --init
```

1.3 Compiling EDK2 Build Tools

- **Compile C Programs**: **EDK2** utilizes a combination of Python scripts and C programs located in the **BaseTools** directory.

  ```
  make -C BaseTools
  ```

  This command builds tools and libraries under **BaseTools/Source/C/bin** and **BaseTools/Source/C/libs**.

- **Tool Wrappers and User Manuals**:
  - **EDK2** offers comprehensive wrappers for the use of build tools on both **Linux** and **Windows** platforms. For detailed guidance, refer to the corresponding directories within the **EDK2** repository.
  - User manuals for various build tools are located under `BaseTools/UserManuals`.

1.4 Building EDKII

- **Initialize Environment**: Source the `edksetup.sh` script:

  ```
  . edksetup.sh
  ```

- **Building EDKII**: Use the `build` command to compile EDKII packages.

1.5 Compiling OVMF (Open Virtual Machine Firmware)

- **Building OVMF**: Execute the following command to build OVMF:

  ```
  build --platform=OvmfPkg/OvmfPkgX64.dsc --arch=X64 --buildtarget=
      RELEASE --tagname=GCC5
  ```

  Note: GCC version 5 or higher is required.

- **Locating Build Artifacts**: The build artifacts can be found in the directory structure: `Build/{Platform Name}/{TARGET}_{TOOL_CHAIN_TAG}/FV`.

1.6 Testing with QEMU

- **Install QEMU**: To install QEMU on your system, run the following command in the terminal:

  ```
  sudo apt-get install qemu-system-x86_64
  ```

- **Running OVMF in QEMU**: To run OVMF with QEMU, use the following command:

```
qemu-system-x86_64 -drive if=pflash,format=raw,readonly,file=Build/
    OvmfX64/RELEASE_GCC5/FV/OVMF_CODE.fd \-drive if=pflash,format=
    raw,file=Build/OvmfX64/RELEASE_GCC5/V/OVMF_VARS.fd \-nographic
    \-net none
```

2 Command Processing Details (HelloWorld.c)

2.1 Commands

- **Quit:** When the user types `"quit"`, the program uses the `Print` function to output a farewell message and then terminates by returning `EFI_SUCCESS`. This is a standard way in UEFI applications to indicate successful completion.
- **List (ls):**
  - The `"ls"` command without flags calls `PrintDirectoryInfo` with a flag value of 0, which likely lists all items in the current directory.
  - For `"ls -n"` and `"ls -s"`, the program checks the flag character following `"ls -"` and calls `PrintDirectoryInfo` with different flag values (1 for 'n' and 2 for 's'). These flags presumably modify the listing behavior, perhaps altering sorting order or information detail level.
- **Remove (rm):** This command uses `StrnCmp` to compare the input buffer with `"rm "`, indicating a removal command. It then calls `OpenFileOrDir` to attempt to open the specified file or directory. If successful, `DeleteFileOrDir` is called to delete it.
- **Make Directory (mkdir):** This command checks if a directory with the specified name already exists using `isThereFile`. If it does not exist, `MakeDirectory` is called, which likely creates a new directory with the given name.
- **Change Directory (cd):** It uses `OpenDirectory` to try and open the directory specified after the `"cd"` command. If the directory is successfully opened, `Current_Dir` is updated to this new directory.
- **Nano:** This is a simple file editing functionality. The code first opens the file in read-only mode to check if it exists. If the file exists, it allows the user to read and potentially modify its content.
- **File Info (info):** This command opens a file in read-only mode and, if successful, calls `PrintFileInfo` to display its details.

- **Concatenate (cat):** Similar to "info", it opens a file and, upon success, reads its content and prints it to the screen.
- **Copy and Cut:** These commands use OpenFileOrDir to locate and open the specified file or directory. The Copy_Cut_Flag is set to 1 for copy and 2 for cut, indicating the action to be taken later (like in the "paste" command).
- **Paste:** This command checks the Copy_Cut_Flag. If a file or directory has been copied or cut, it uses CopyDirectoryRecursive to paste it into the current directory. If the Copy_Cut_Flag was 2 (cut), the original file/directory is deleted after pasting.
- **Help:** Likely displays a list of available commands and their usage.
- **Search:** It appears to search for files matching a given pattern and prints the results.
- **Invalid Command:** If a command is not recognized, it prints an error message and shows help.

2.2 help

```
void help(){
    Print(L"Supported commands are:\n");
    Print(L"ls - list all files and directories in the current
       directory\n");
    Print(L"ls -n - list all files and directories in the current
        directory with alphabetical order\n");
    Print(L"rm <file_name> - remove file or directory\n");
    Print(L"mkdir <directory_name> - create a new directory\n");
    Print(L"cd <directory_name> - change current directory\n");
    Print(L"nano <file_name> - create or edit a file\n");
    Print(L"info <file_name> - print name and size of a file\n");
    Print(L"cat <file_name> - print content of a file\n");
    Print(L"copy <file_name> - copy a file or directory\n");
    Print(L"cut <file_name> - cut a file or directory\n");
    Print(L"paste - paste a file or directory\n");
    Print(L"search <sub_name> - show files and directories that
       contains this substring\n");
    Print(L"quit - exit the program\n");
}
```

- The function help prints a list of supported commands to the console.

  - Each command is accompanied by a brief description of its function, providing a user-friendly guide for interacting with the program.
  - This function is crucial for usability, especially for new users who may not be familiar with the available commands.

- The function `isSubString` checks if a given substring is present within another string.
  - It uses the UEFI library function `StrStr` to search for the substring.
  - If the substring is found, the function returns `TRUE`, otherwise `FALSE`.
  - This function can be useful in commands where pattern matching or search functionality is required, such as the 'search' command listed in the 'help' function.

2.3 GetStringFromIndex

The function `GetStringFromIndex` is designed to read a string input from the console and store it in a buffer.

- Takes a buffer, its size, and a starting index as arguments.
- Reads key strokes using `ReadKeyStroke` and stores them in the buffer, handling delete keys and stopping when the enter key is pressed.
- Supports real-time deletion and echoing of characters.
- Returns `EFI_SUCCESS` on successful completion, and an error status if an error occurs during the key reading process.

2.4 PrintDirectoryInfo

`PrintDirectoryInfo` function takes a directory and a flag as input.

- It initializes two arrays to store names of files and directories.
- The function then reads each entry in the given directory, categorizing them into files and directories based on the result of `is_name_file`.
- If the flag is set to 1, it sorts both files and directories using the `BubbleSortStrings` function.

2.5 OpenFileOrDir

The function `OpenFileOrDir` is designed to open either a file or directory with read and write permissions, without specifying the file type.

- Returns an EFI status to indicate the outcome.

2.6 DeleteFileOrDir

The function `DeleteFileOrDir` takes a pointer to an `EFI_FILE_PROTOCOL` object representing either a file or a directory.

- It first checks if the object is a file using the `is_file` function.
- If it's a file, it directly deletes it.
- If it's a directory, the function iterates through all files and subdirectories within it (skipping the special entries "." and ".."), and recursively calls itself to delete each of them.
- After deleting all contents, it deletes the directory itself.
- The function returns an `EFI_STATUS` value indicating the success or failure of the delete operation.

2.7 MakeDirectory

The function `MakeDirectory` attempts to create a new directory with the given name in the current directory.

- Sets the mode to create, write, and read, specifying the file type as a directory.
- Returns the status of the directory creation attempt.

2.8 OpenDirectory

The function `OpenDirectory` opens a directory for reading and writing.

- Returns the status of the operation, indicating whether the directory was successfully opened or not.

2.9 PrintFileInfo

The function `PrintFileInfo` takes a pointer to an `EFI_FILE_PROTOCOL` object.

- Uses `FileHandleGetInfo` to retrieve file information stored in `FileInfo`.
- Prints the file size in bytes and the file name to the standard output using the `Print` function.
- This function could be extended to print additional file attributes by adding more `Print` statements.

2.10 CopyDirectoryRecursive

The function `CopyDirectoryRecursive` recursively copies files and directories from a source to a destination directory.

- It first checks whether the source is a file or a directory.
- If it's a file, it simply copies it to the destination directory.
- If it's a directory, the function creates a corresponding directory in the destination and then iterates through all the contents of the source directory.
- For each item in the source directory, it recursively calls itself to copy the item (be it a file or a subdirectory) to the newly created directory in the destination.
- This recursive approach ensures that the entire directory structure and its contents are copied.
- The function returns an `EFI_STATUS` to indicate the success or failure of the operation.

2.11 is_file

The `is_file` function is used to determine whether the provided `EFI_FILE_PROTOCOL` object represents a file or a directory.

- It utilizes the `FileHandleGetInfo` function to retrieve file information.
- If the `Attribute` field of the `FileInfo` structure contains the `EFI_FILE_DIRECTORY` flag, it indicates a directory, and the function returns 0.
- Otherwise, it returns 1, indicating that the object represents a file.

2.12 PrintGetPosition

The `PrintGetPosition` function accepts an `EFI_FILE_HANDLE` object as its input.

- It retrieves the current position within the file using the `FileHandleGetPosition` function.
- Subsequently, it stores this position in the variable `Result` and proceeds to print it.
- This function is particularly useful for determining the read/write position within the file.

2.13 GetString

The `GetString` function is a straightforward wrapper around `GetStringFromIndex`.

- It initializes string reading from the beginning of the buffer (index 0).
- This function is useful for obtaining a new string input.

2.14 OpenOnlyRead

The `OpenOnlyRead` function opens a directory or file with read-only permissions.

- It is used when the intention is to check for existence or perform operations that do not require write permissions.

2.15 isThereFile

The `isThereFile` function checks whether a file or directory with the specified name exists in the current directory.

- This function uses `OpenOnlyRead` for this purpose.
- It returns `TRUE` if the file or directory exists; otherwise, it returns `FALSE`.

2.16 OpenFileOnlyRead

The `OpenFileOnlyRead` function is utilized to open a file with both read and write permissions, while specifying the file type as an archive.

- The status returned indicates whether the operation was successful or not.

2.17 OpenFileCreate

The `OpenFileCreate` function is employed to create a new file with both read and write permissions, classifying it as an archive.

- This function returns an EFI status that indicates the success or failure of the file creation operation.

2.18 GetSize

The `GetSize` function calculates the size of a string in bytes, including the null terminator.

- It iterates through each character until it encounters the null character and then returns the size in bytes.

2.19 is_name_file

The `is_name_file` function verifies whether a provided name corresponds to a file in the current directory.

- It attempts to open the file or directory and employs `is_file` to determine if it's a file. If the opening fails, it returns -1.

2.20 PrintFiles

The `PrintFiles` function iterates through an array of file names and prints each one, adding the prefix "file: ".

- This function is handy for displaying a list of files.

2.21 PrintDirectories

Similar to `PrintFiles`, the `PrintDirectories` function prints each directory name in an array, with the prefix "directory: ".

2.22 String Manipulation and Sorting

These functions are used for string manipulation and sorting.

- `StringCompare` compares two strings.
- `SwapStrings` swaps two string pointers.
- `BubbleSortStrings` implements the bubble sort algorithm to sort an array of strings based on their lexicographical order.

2.23 CopyFile

The `CopyFile` function accepts two `EFI_FILE_PROTOCOL` objects as parameters: one representing the source file and the other representing the destination file.

- Its purpose is to copy the contents from the source file to the destination file.
- This is achieved by reading data in chunks from the source file and subsequently writing that data to the destination file.
- The process iterates until all data is copied or an error occurs.
- This function is designed to handle files of any size by processing them in manageable chunks.
- It returns an `EFI_STATUS` to indicate the outcome of the copy operation.

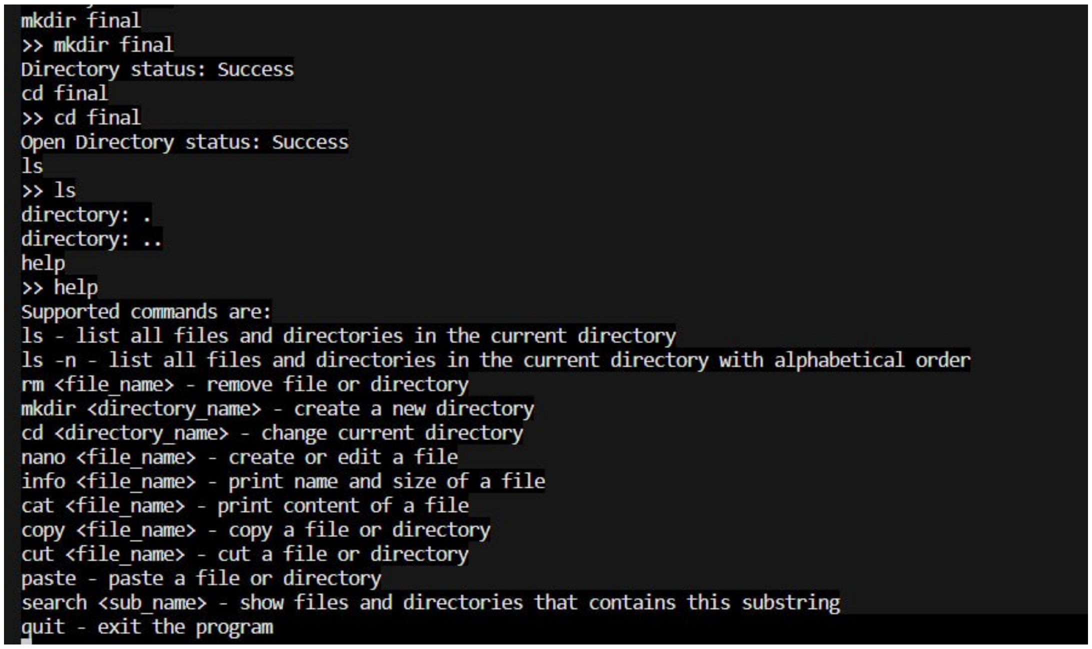
2.24 SearchFiles

The `SearchFiles` function is designed to search for files and directories within the `Current_Dir` whose names contain the substring `file_name`.

- It maintains a list of paths for the found items in the `founded_files` array and keeps count of the number of found items in `count`.
- The function iterates over each item in `Current_Dir`, excluding special directories `"."` and `".."`.
- For each item, it checks if its name contains the specified substring. If a match is found, the path is added to the `founded_files` array.
- If the item is a directory (not a file), the function recursively calls itself to search within that directory.

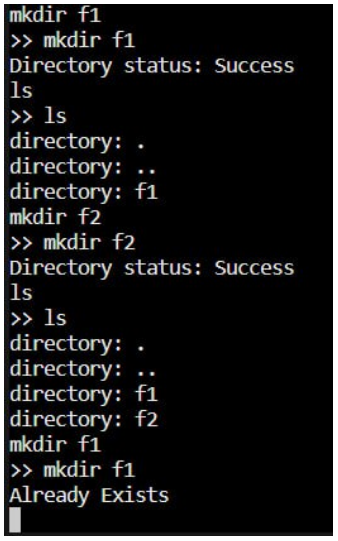
- This recursive approach allows for a thorough search through all subdirectories.
- The function handles path concatenation and memory allocation for new paths.
- It returns an `EFI_STATUS` to indicate the success or failure of the search operation.

2.25 Check Test Results



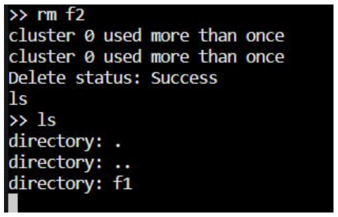
```
mkdir final
>> mkdir final
Directory status: Success
cd final
>> cd final
Open Directory status: Success
ls
>> ls
directory: .
directory: ..
help
>> help
Supported commands are:
ls - list all files and directories in the current directory
ls -n - list all files and directories in the current directory with alphabetical order
rm <file_name> - remove file or directory
mkdir <directory_name> - create a new directory
cd <directory_name> - change current directory
nano <file_name> - create or edit a file
info <file_name> - print name and size of a file
cat <file_name> - print content of a file
copy <file_name> - copy a file or directory
cut <file_name> - cut a file or directory
paste - paste a file or directory
search <sub_name> - show files and directories that contains this substring
quit - exit the program
```

Figure 1: help



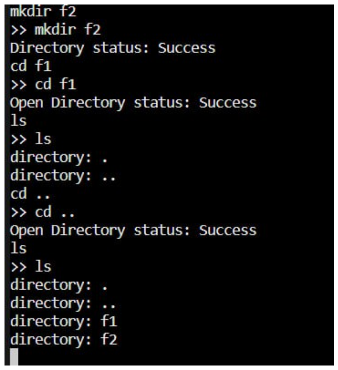
```
mkdir f1
>> mkdir f1
Directory status: Success
ls
>> ls
directory: .
directory: ..
directory: f1
mkdir f2
>> mkdir f2
Directory status: Success
ls
>> ls
directory: .
directory: ..
directory: f1
directory: f2
mkdir f1
>> mkdir f1
Already Exists
```

Figure 2: mkdir



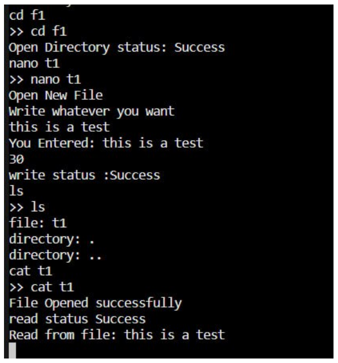
```
>> rm f2
cluster 0 used more than once
cluster 0 used more than once
Delete status: Success
ls
>> ls
directory: .
directory: ..
directory: f1
```

Figure 3: rm



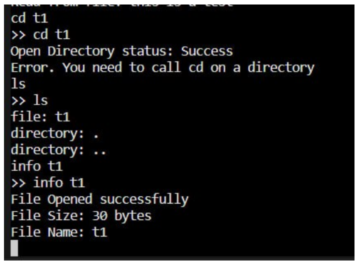
```
mkdir f2
>> mkdir f2
Directory status: Success
cd f1
>> cd f1
Open Directory status: Success
ls
>> ls
directory: .
directory: ..
cd ..
>> cd ..
Open Directory status: Success
ls
>> ls
directory: .
directory: ..
directory: f1
directory: f2
```

Figure 4: mkdir



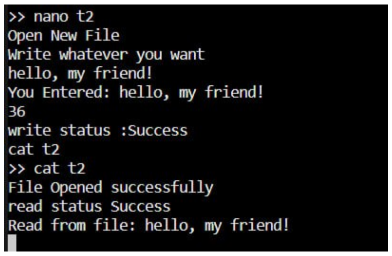
```
cd f1
>> cd f1
Open Directory status: Success
nano t1
>> nano t1
Open New File
Write whatever you want
this is a test
You Entered: this is a test
30
write status :Success
ls
>> ls
file: t1
directory: .
directory: ..
cat t1
>> cat t1
File Opened successfully
read status Success
Read from file: this is a test
```

Figure 5: nano, cat



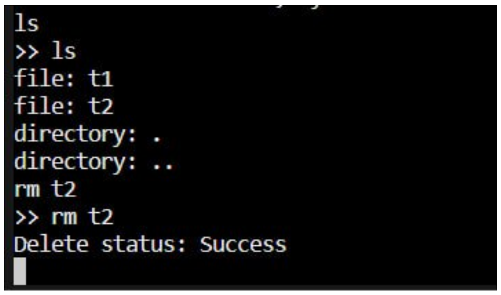
```
cd t1
>> cd t1
Open Directory status: Success
Error. You need to call cd on a directory
ls
>> ls
file: t1
directory: .
directory: ..
info t1
>> info t1
File Opened successfully
File Size: 30 bytes
File Name: t1
```

Figure 6: info



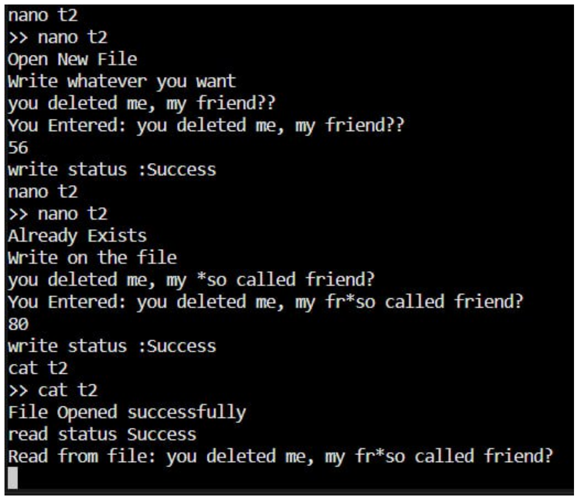
```
>> nano t2
Open New File
Write whatever you want
hello, my friend!
You Entered: hello, my friend!
36
write status :Success
cat t2
>> cat t2
File Opened successfully
read status Success
Read from file: hello, my friend!
```

Figure 7: nano, cat



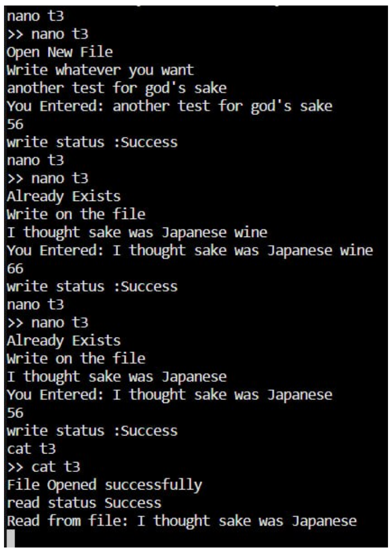
```
ls
>> ls
file: t1
file: t2
directory: .
directory: ..
rm t2
>> rm t2
Delete status: Success
```

Figure 8: rm



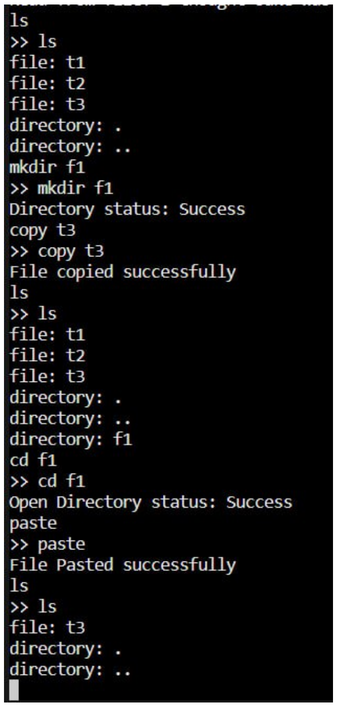
```
nano t2
>> nano t2
Open New File
Write whatever you want
you deleted me, my friend??
You Entered: you deleted me, my friend??
56
write status :Success
nano t2
>> nano t2
Already Exists
Write on the file
you deleted me, my *so called friend?
You Entered: you deleted me, my fr*so called friend?
80
write status :Success
cat t2
>> cat t2
File Opened successfully
read status Success
Read from file: you deleted me, my fr*so called friend?
```

Figure 9: nano, cat



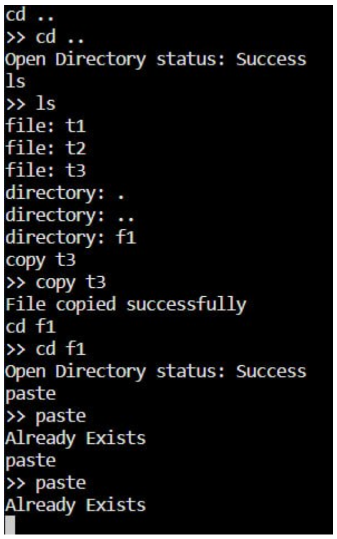
```
nano t3
>> nano t3
Open New File
Write whatever you want
another test for god's sake
You Entered: another test for god's sake
56
write status :Success
nano t3
>> nano t3
Already Exists
Write on the file
I thought sake was Japanese wine
You Entered: I thought sake was Japanese wine
66
write status :Success
nano t3
>> nano t3
Already Exists
Write on the file
I thought sake was Japanese
You Entered: I thought sake was Japanese
56
write status :Success
cat t3
>> cat t3
File Opened successfully
read status Success
Read from file: I thought sake was Japanese
```

Figure 10: nano, cat



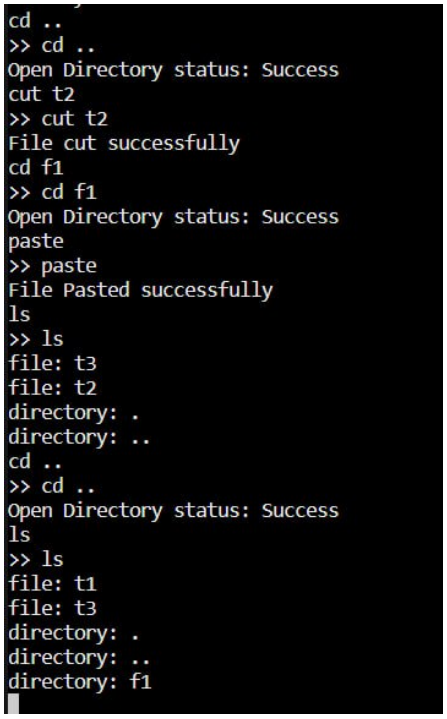
```
ls
>> ls
file: t1
file: t2
file: t3
directory: .
directory: ..
mkdir f1
>> mkdir f1
Directory status: Success
copy t3
>> copy t3
File copied successfully
ls
>> ls
file: t1
file: t2
file: t3
directory: .
directory: ..
directory: f1
cd f1
>> cd f1
Open Directory status: Success
paste
>> paste
File Pasted successfully
ls
>> ls
file: t3
directory: .
directory: ..
```

Figure 11: copy, paste



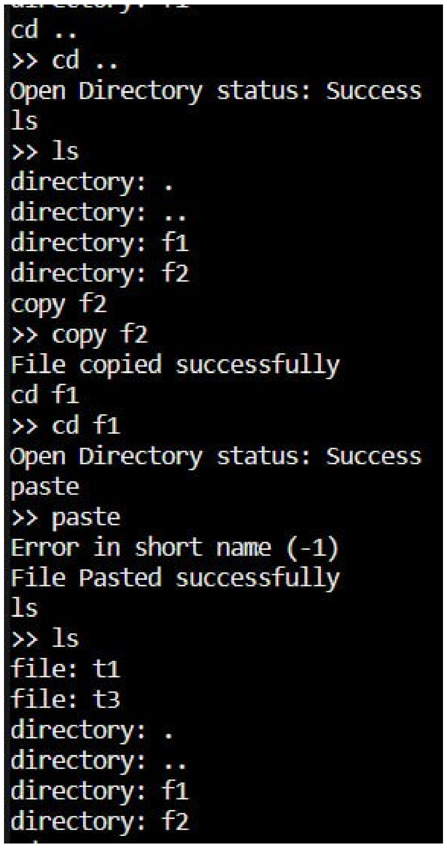
```
cd ..
>> cd ..
Open Directory status: Success
ls
>> ls
file: t1
file: t2
file: t3
directory: .
directory: ..
directory: f1
copy t3
>> copy t3
File copied successfully
cd f1
>> cd f1
Open Directory status: Success
paste
>> paste
Already Exists
paste
>> paste
Already Exists
```

Figure 12: nano, cat



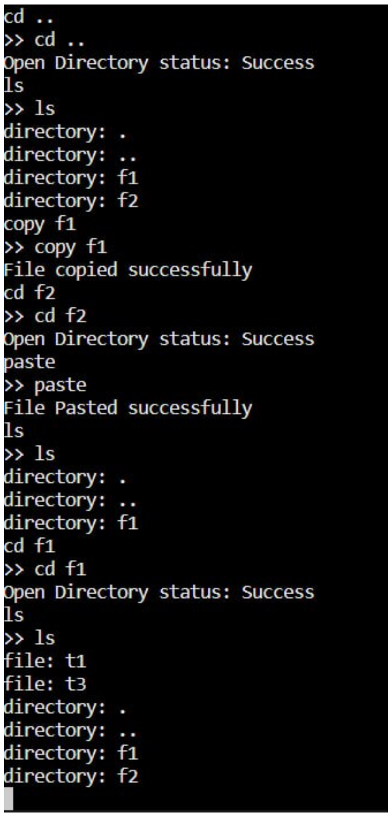
```
cd ..
>> cd ..
Open Directory status: Success
cut t2
>> cut t2
File cut successfully
cd f1
>> cd f1
Open Directory status: Success
paste
>> paste
File Pasted successfully
ls
>> ls
file: t3
file: t2
directory: .
directory: ..
cd ..
>> cd ..
Open Directory status: Success
ls
>> ls
file: t1
file: t3
directory: .
directory: ..
directory: f1
```

Figure 13: cut, paste



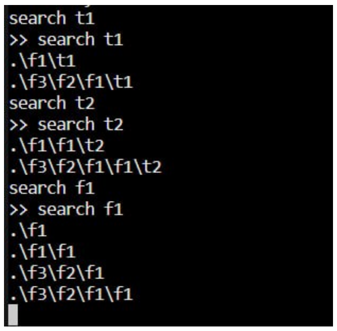
```
cd ..
>> cd ..
Open Directory status: Success
ls
>> ls
directory: .
directory: ..
directory: f1
directory: f2
copy f2
>> copy f2
File copied successfully
cd f1
>> cd f1
Open Directory status: Success
paste
>> paste
Error in short name (-1)
File Pasted successfully
ls
>> ls
file: t1
file: t3
directory: .
directory: ..
directory: f1
directory: f2
```

Figure 14: copy, paste



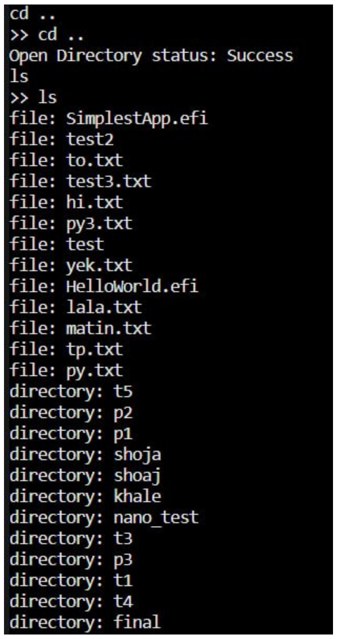
```
cd ..
>> cd ..
Open Directory status: Success
ls
>> ls
directory: .
directory: ..
directory: f1
directory: f2
copy f1
>> copy f1
File copied successfully
cd f2
>> cd f2
Open Directory status: Success
paste
>> paste
File Pasted successfully
ls
>> ls
directory: .
directory: ..
directory: f1
cd f1
>> cd f1
Open Directory status: Success
ls
>> ls
file: t1
file: t3
directory: .
directory: ..
directory: f1
directory: f2
```

Figure 15: copy, paste



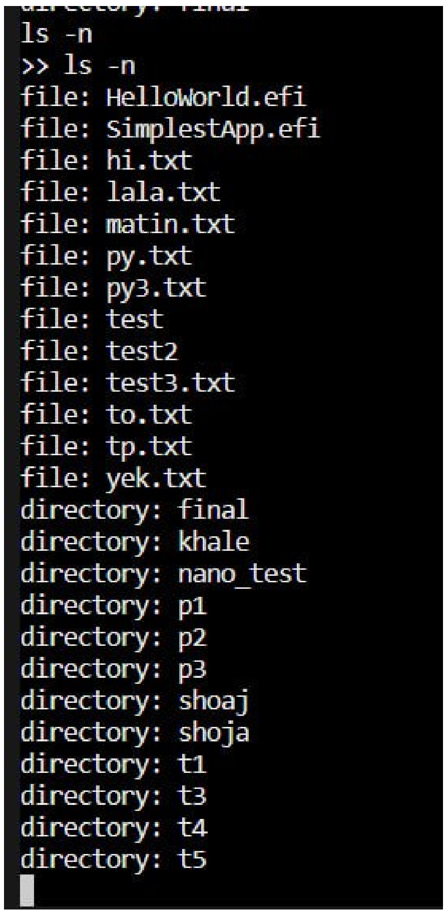
```
search t1
>> search t1
.\f1\t1
.\f3\f2\f1\t1
search t2
>> search t2
.\f1\f1\t2
.\f3\f2\f1\f1\t2
search f1
>> search f1
.\f1
.\f1\f1
.\f3\f2\f1
.\f3\f2\f1\f1
```

Figure 16: search



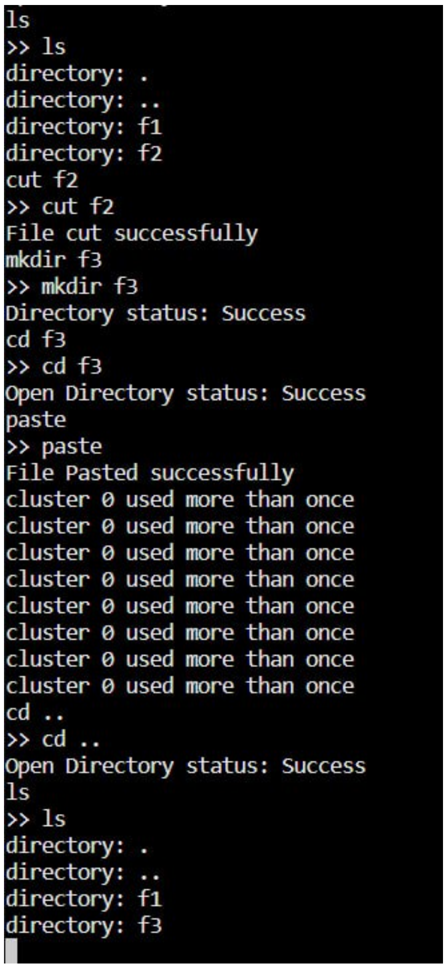
```
cd ..
>> cd ..
Open Directory status: Success
ls
>> ls
file: SimplestApp.efi
file: test2
file: to.txt
file: test3.txt
file: hi.txt
file: py3.txt
file: test
file: yek.txt
file: HelloWorld.efi
file: lala.txt
file: matin.txt
file: tp.txt
file: py.txt
directory: t5
directory: p2
directory: p1
directory: shoja
directory: shoaj
directory: khale
directory: nano_test
directory: t3
directory: p3
directory: t1
directory: t4
directory: final
```

Figure 17: ls



```
ls -n
>> ls -n
file: HelloWorld.efi
file: SimplestApp.efi
file: hi.txt
file: lala.txt
file: matin.txt
file: py.txt
file: py3.txt
file: test
file: test2
file: test3.txt
file: to.txt
file: tp.txt
file: yek.txt
directory: final
directory: khale
directory: nano_test
directory: p1
directory: p2
directory: p3
directory: shoaj
directory: shoja
directory: t1
directory: t3
directory: t4
directory: t5
```

Figure 18: ls -n



```
ls
>> ls
directory: .
directory: ..
directory: f1
directory: f2
cut f2
>> cut f2
File cut successfully
mkdir f3
>> mkdir f3
Directory status: Success
cd f3
>> cd f3
Open Directory status: Success
paste
>> paste
File Pasted successfully
cluster 0 used more than once
cluster 0 used more than once
cluster 0 used more than once
cluster 0 used more than once
cluster 0 used more than once
cluster 0 used more than once
cluster 0 used more than once
cluster 0 used more than once
cd ..
>> cd ..
Open Directory status: Success
ls
>> ls
directory: .
directory: ..
directory: f1
directory: f3
```

Figure 19: cut, mkdir, paste, ls

3 UEFI application Configuration (HelloWorld.inf)

3.1 Defines

Sets constants and configurations for the build process.

- `INF_VERSION`: Version of the INF file format.
- `BASE_NAME`: Base name of the application, "HelloWorld".
- `FILE_GUID`: Globally unique identifier for the file.
- `MODULE_TYPE`: Type of the module, a UEFI application.
- `VERSION_STRING`: Application version.
- `ENTRY_POINT`: Entry function of the application, `UefiMain`.

3.2 Sources

Lists the source code file.

- `HelloWorld.c`: C language file containing the application source code.

3.3 Packages

Specifies required packages.

- `MdePkg/MdePkg.dec`: Package providing common UEFI library classes.

3.4 LibraryClasses

Lists used library classes.

- `UefiApplicationEntryPoint`: Provides the entry point for UEFI applications.
- `UefiLib`: Basic UEFI functions.
- `UefiFileHandleLib`: Working with UEFI file handles.
- `MemoryAllocationLib`: Memory allocation tasks.

3.5 Protocols

Lists UEFI protocols used by the application.

- `gEfiSimpleFileSystemProtocolGuid`: Protocol identifier for UEFI's Simple File System Protocol.

This configuration file sets up a UEFI application named "HelloWorld", detailing its dependencies, source files, and required libraries. The application begins execution from the `UefiMain` function in `HelloWorld.c` and utilizes specific UEFI protocols and libraries for file system and memory operations.

4 UEFI development Configuration (UefiLessonsPkg.dsc)

4.1 Defines

- `DSC_SPECIFICATION`: Version of the DSC file format.
- `PLATFORM_GUID`: Unique identifier for the platform.
- `PLATFORM_VERSION`: Version number of the platform.
- `PLATFORM_NAME`: Name of the package, "UefiLessonsPkg".
- `SKUID_IDENTIFIER`: SKU identifier, set to "DEFAULT".
- `SUPPORTED_ARCHITECTURES`: CPU architecture, here X64.
- `BUILD_TARGETS`: Build configuration, set to "RELEASE".

4.2 LibraryClasses

Lists the library classes with their paths. Examples include:

- `UefiApplicationEntryPoint`, `UefiBootServicesTableLib`, `DebugLib`, etc., from the `MdePkg`.

4.3 Components

Specifies UEFI applications or drivers in the package:

- `UefiLessonsPkg/SimplestApp/SimplestApp.inf`
- `UefiLessonsPkg/HelloWorld/HelloWorld.inf`

This file is a project configuration for UEFI development, specifying build settings, libraries, and components necessary to build UEFI applications within the "UefiLessonsPkg" package, targeting 64-bit architecture for a release build. It includes components like "SimplestApp" and "HelloWorld" and a variety of libraries for UEFI development functionalities.

5 Related Links

- `https://github.com/tianocore/edk2`
- `https://uefi.org/specs/UEFI/2.10/13_Protocols_Media_Access.html`
- `https://github.com/Kostr/UEFI-Lessons`

6 Authors

- Mohammadreza AhmadiTeshnizi
- Ali Abbasi
- Matin Shoja: