Cykor Assignment 2: Simulating Linux Shell with C++

Jinho Kim

Department of Cyber Defence, Korea University
ID: 2025350219
jhk0317 at korea dot ac dot kr

May 12, 2025

Contents

I Introduction

This assignment aims to simulate the behaviour of the Linux shell. The program successfully supports shell commands such as `mkdir`, `ls`, `cd`, and `pwd`, et cetera. Moreover, it replicates key aspects of actual shell functionality, including the use of `fork()` for process creation, background execution using the `&` operator, and pipeline handling via the `|` symbol.

The program is fully written in C++. Although the program could have been written in classic C as well, this project utilises unique C++ functionalities such as classes and constructors, easier string handling, modern syntax of using `nullptr` over classic `NULL`, usage of member functions and variables using the `->` operator, and so on. The program follows the modern convention of C++. The code refrains from the usage of default namespace declaration, and actively uses the OOP (Object Oriented Programming) aspects of the C++ programming language.

The program is separated into 6 main modules, `Directory`, `CommandParser`, `CommandRunner`, `PipelineRunner`, `MainRunner`, and the driver `main` function. In this report, we will go over all 6 parts of the method. Headerfiles will not be reviewed, but can be viewed in the GitHub repository, here.

II `main.cpp`

```
1  #include "Directory.h"
2  #include "MainRunner.h"
3
4  Directory* root              = new Directory("/");
5  Directory* currentDir        = root;
6  std::string username         = "your_name";
7  std::string devicename       = "your_device_name";
8
9  int main() {
10     system("clear");
11
12     while (true) {
13         MainRunner();
14     }
15
16     return 0;
17 }
```

In the `main.cpp` file, no special functionalities are provided. All the input handlings and actual processes are done in other modules. In this file, 2 headerfiles are included. These headerfiles are written in the same directory, and handles most of the logics.

The `Directory.h` headerfile is in charge of declaring and handling the user-defined class `Directory`, and must be included since the root directory and the current directory pointer variables are declared at lines 4 and 5. Further explanations on the `Directory` class are provided in Section III.

The `MainRunner.h` headerfile is in charge of handling user inputs and executing given commands. The `MainRunner()` method under the `MainRunner.h` file is run inside the loop under the `main` function.

In the main method, `system("clear")` method is used to clear out existing shell information,

which is only for the user interface and experience.

Line 12-14 simply runs the `MainRunner()` method infinitely until the `exit` command is given. Since the command is handled inside the `MainRunner()` method, here, the loop is simply designed to run infinitely.

Note that the four variables declared at line 4-7 are used in other files as well, as a shared external variable.

III `Directory` class

III.1 `Directory.h`

```
1  #pragma once
2  #include <string>
3
4  class Directory {
5  public:
6      std::string     dirname;
7      Directory*      parent;
8      Directory*      subdir;
9      Directory*      siblingdir;
10
11     Directory(std::string name);
12
13     void        mkdir(Directory* currentDir, std::string name);
14     void        listDir(bool isBackground = false);
15     void        printWorkingDir(Directory* currentDir, bool isBackground = false);
16     bool        changeDir(Directory*& currentDir, const std::string& name);
17     Directory*  findSubdir(std::string name);
18
19 private:
20 protected:
21 };
```

The `Directory.h` file declares the structure of the `Directory` class. This class represents a single directory inside the shell environment, which is later used in `mkdir`, `pwd`, `cd`, and much more commands that are implemented.

The `Directory` class has four member variables, to identify each directory. The `dirname` variable represents the name of the directory. It is used to distinguish the directory, but not actively used in the logic process. Most of the program logic is handled mostly through the instance's pointer members. The `parent` variable holds the pointer of its parent directory.

The `subdir` is where it gets complex, since a directory may hold more than one subdirectory. Instead of holding all subdirectories as a pointer, this program only holds the first subdirectory created, and the rest of the subdirectories are handled via the next member variable, the `siblingdir`. When the parent directory's `subdir` variable is already occupied (in other words, not `null`), it automatically finds one of the subdirectories where its `siblingdir` is not occupied. Hence, it is possible to handle all the subdirectories without actually storing all of their pointers. The detailed explanation about this logic will be provided in Section III.2.

Next is followed by the constructor. The only parameter required for the constructor is the name in string type, which will be used to initialize the `dirname` variable on the instantiation

process.

There are five methods inside the `Directory` class, `mkdir()`, `listDir()`, `printWorkingDir()`, `changeDir()`, `findSubdir()`, which is for `mkdir`, `ls`, `pwd`, `cd` command, respectively, and a supporting function `findSubdir()`.

All the members inside the class is declared publicly by default for simplicity, but since there may be concerns on possible vulnerability, it may be better to implement accessor and mutator methods in the future.

III.2 `Directory.cpp`

```cpp
#include "Directory.h"

#include <iostream>
#include <vector>
#include <sstream>
#include <unistd.h>
#include <sys/wait.h>

Directory::Directory(std::string name) {
    dirname = name;
    parent = nullptr;
    subdir = nullptr;
    siblingdir = nullptr;
}

void Directory::mkdir(Directory* currentDir, std::string name) {
    Directory* newDir = new Directory(name);
    newDir->parent = currentDir;

    if (currentDir->subdir == nullptr) {
        currentDir->subdir = newDir;
    } else {
        Directory* temp = currentDir->subdir;
        while (temp->siblingdir != nullptr) {
            temp = temp->siblingdir;
        }
        temp->siblingdir = newDir;
    }
}

void Directory::listDir(bool isBackground) {
    pid_t pid = fork();

    if (pid < 0) {
        std::cerr << "Error! Fork failed for ls." << std::endl;
        return;
    }

    if (pid == 0) {
        Directory* temp = subdir;
        while (temp != nullptr) {
            std::cout << temp->dirname << "  ";
            temp = temp->siblingdir;
```

```
        }
        std::cout << std::endl;
        _exit(0);
    } else {
        if (!isBackground) {
            int status;
            waitpid(pid, &status, 0);
        } else {
            std::cout << "Background process ls started with PID: "
            << pid << std::endl;
        }
    }
}

Directory* Directory::findSubdir(std::string name) {
    Directory* temp = subdir;
    while (temp != nullptr) {
        if (temp->dirname == name) {
            return temp;
        }
        temp = temp->siblingdir;
    }
    return nullptr;
}

bool Directory::changeDir(Directory*& currentDir, const std::string& dirname) {
    std::stringstream ss(dirname);
    std::string token;
    Directory* temp = currentDir;

    while (std::getline(ss, token, '/')) {
        if (token.empty() || token == ".") {
            continue;
        } else if (token == "..") {
            if (temp->parent != nullptr) {
                temp = temp->parent;
            } else {
                std::cout << "Already at root directory" << std::endl;
                return false;
            }
        } else {
            Directory* nextDir = temp->findSubdir(token);
            if (nextDir != nullptr) {
                temp = nextDir;
            } else {
                std::cout << "Directory not found: " << token << std::endl;
                return false;
            }
        }
    }
    currentDir = temp;
    return true;
}

void Directory::printWorkingDir(Directory* currentDir, bool isBackground) {
    pid_t pid = fork();

    if (pid < 0) {
```

```
        std::cerr << "Error! Fork failed for pwd." << std::endl;
        return;
    }

    if (pid == 0) {
        std::vector<std::string> path;
        Directory* temp = currentDir;

        while (temp != nullptr) {
            path.push_back(temp->dirname);
            temp = temp->parent;
        }

        std::cout << "/";
        for (std::vector<std::string>::reverse_iterator it = path.rbegin();
            it != path.rend(); ++it) {
            if (*it != "/") std::cout << *it << "/";
        }
        std::cout << std::endl;

        _exit(0);
    } else {
        if (!isBackground) {
            int status;
            waitpid(pid, &status, 0);
        } else {
            std::cout << "Background process pwd started with PID: "
            << pid << std::endl;
        }
    }
}
```

Since the `Directory.cpp` file contains multiple parts, the report will be written part-by-part.

III.2.1 Standard Libraries

```
#include "Directory.h"

#include <iostream>
#include <vector>
#include <sstream>
#include <unistd.h>
#include <sys/wait.h>
```

The `Directory.h` header is included to implement the pre-declared `Directory` class, which was explained in III.1.

The `iostream` standard library is required for standard printing functions, in this code, `cout` and `cerr` (for error handlings).

The `vector` standard library is used to deal with the directory structure inside the `printWorkingDirectory()` function, since it requires reverse traversing from the bottom-most directory to the root directory. This behaviour will be better implemented using the `vector` data structure, compared to plain array.

The `sstream` standard library is used in the `changeDir()` function, as it simplifies the processing of multiple tokens when a relative directory path contains several components.

Both `unistd.h` and `sys/wait.h` standard libraries are used in handling processes. `fork()` function requires `unistd.h`, and `waitpid()` function requires `sys/wait.h`.

III.2.2 Constructor `Directory()`

```
Directory::Directory(std::string name) {
    this->dirname = name;
    this->parent = nullptr;
    this->subdir = nullptr;
    this->siblingdir = nullptr;
}
```

In this class, the constructor is not used outside the class. Therefore, there are no need to assign parent directory pointer value on instantiation. Hence, the constructor only takes the directory name, and assigns `nullptr` to all other member variables.

Again, for clarification, the constructor is only used inside the `mkdir()` function, which is located inside the class.

III.2.3 Method `mkdir()`

```
void Directory::mkdir(Directory* currentDir, std::string name) {
    Directory* newDir = new Directory(name);
    newDir->parent = currentDir;

    if (currentDir->subdir == nullptr) {
        currentDir->subdir = newDir;
    } else {
        Directory* temp = currentDir->subdir;
        while (temp->siblingdir != nullptr) {
            temp = temp->siblingdir;
        }
        temp->siblingdir = newDir;
    }
}
```

The `mkdir()` method takes the pointer to the current directory and the name of the new directory as a parameter. First, by using the constructor mentioned above at section III.2.2, we instantiate the new directory. Then the new directory's parent directory is assigned as to current directory.

Now, to specify the subdirectory relationship, we first check if the current directory's subdirectory is not assigned yet. In other words, if the directory that we just made is the first subdirectory of the current directory, we simply assign the new directory to the current directory's `subdir` variable, specifing the relationship.

If there already exists another subdirectory under the current directory, we need to search for the nearest subdirectory whose sibling directory variable is still empty. By iterating through the while loop until it's found, we can successfully search the empty sibling directory variable. Once

found, we assign the sibling directory variable to the directory we just created. Note that we use the `nullptr` constant literal over the classic `NULL`, following the C++ convention. (`nullptr` is available in C++11 or later) The usage of `nullptr` will be kept throughout the whole codebase.

However, there are some limitations on this method. While the actual Linux shell checks if the given directory name already exists, this logic does not go through those checks. This can be done simply with another recursive method checking through the directory name, but it is not implemented here for simplicity.

III.2.4 Method `listDir()`

```
1  void Directory::listDir(bool isBackground) {
2      pid_t pid = fork();
3
4      if (pid < 0) {
5          std::cerr << "Error! Fork failed for ls." << std::endl;
6          return;
7      }
8
9      if (pid == 0) {
10         Directory* temp = subdir;
11         while (temp != nullptr) {
12             std::cout << temp->dirname << "  ";
13             temp = temp->siblingdir;
14         }
15         std::cout << std::endl;
16         _exit(0);
17     } else {
18         if (!isBackground) {
19             int status;
20             waitpid(pid, &status, 0);
21         } else {
22             std::cout << "Background process ls started with PID: "
23             << pid << std::endl;
24         }
25     }
26 }
```

The `listDir()` function behaves slightly different from the above `mkdir()` function. Since the Linux shell creates a child process when the `ls` command is given, this `listDir()` method aims to copy the actual mechanism of it as well.

When the `listDir()` method is called, we execute the `fork()` function, creating a fork. Then we check if `pid` $<$ `0`, which means if there is a problem with fork, the program returns an error.

If we can confirm that the `fork()` function is executed properly without an error, we now go through the actual directory listing process.

Just like the `mkdir()` function above, we iterate through the sibling directories until a directory contains no sibling directory, and printing out the directory name during every iteration.

At line 16, the function calls `_exit(0)`. Unlike `exit()`, `_exit()` terminates the process immediately without performing standard clean-up tasks such as flushing I/O buffers or calling object destructors. [1] This avoids duplicated clean-up operations, which are the responsibility

of the parent process.

As we take a look at line 14 of the `Directory.h` headerfile,

```
    void        listDir(bool isBackground = false);
```

we can see that the parameter `isBackground` is false by default, that is, if no parameter is given specifically, the `listDir()` function is not run in background by default.

When the parameter is not given, it means that the process does not run in background. In this case, the shell does not invoke `waitpid()` function, and the parent process continues execution without waiting for the child process to terminate. This behaviour mimics that of a real Linux shell, where appending an ampersand (`&`) to a command runs it in the background.

III.2.5 Method `findSubir()`

```
1  Directory* Directory::findSubdir(std::string name) {
2      Directory* temp = subdir;
3      while (temp != nullptr) {
4          if (temp->dirname == name) {
5              return temp;
6          }
7          temp = temp->siblingdir;
8      }
9      return nullptr;
10 }
```

As we take a look at the `findSubdir()` function, we can see that it is responsible for searching a subdirectory within the current directory by matching the given string parameter `name` and the `dirname` variable in the traversing `Directory` instance.

The function begins by declaring a temporary pointer `temp`, which is initially assigned to `subdir`, representing the first subdirectory of the current directory. It then enters a `while` loop that continues as long as `temp` is not `nullptr`, indicating there are still sibling directories left to examine.

Within the loop, the function compares the `dirname` of the current subdirectory pointed to by `temp` with the given `name`. If a match is found, the function immediately returns that directory's pointer. Otherwise, it advances to the next sibling directory by updating `temp` to `temp->siblingdir`.

If the loop completes without finding a match, this indicates that no subdirectory with the given name exists in the current directory. In this case, the function returns `nullptr`.

In summary, `findSubdir()` performs a search among the current directory's immediate subdirectories, returning the matching directory if found, and `nullptr` otherwise. This function is used in `changeDir()` function, which implements the `cd` command.

Note that this `findSubdir()` function is not used on its own. It is only a supporting function that is not called outside the class. It is possible to declare this function private.

III.2.6 Method `changeDir()`

```
bool Directory::changeDir(Directory*& currentDir, const std::string& dirname) {
    std::stringstream ss(dirname);
    std::string token;
    Directory* temp = currentDir;

    while (std::getline(ss, token, '/')) {
        if (token.empty() || token == ".") {
            continue;
        } else if (token == "..") {
            if (temp->parent != nullptr) {
                temp = temp->parent;
            } else {
                std::cout << "Already at root directory" << std::endl;
                return false;
            }
        } else {
            Directory* nextDir = temp->findSubdir(token);
            if (nextDir != nullptr) {
                temp = nextDir;
            } else {
                std::cout << "Directory not found: " << token << std::endl;
                return false;
            }
        }
    }
    currentDir = temp;
    return true;
}
```

The `changeDir()` method allows for directory traversal within the simulated file system by modifying the `currentDir` reference according to the path provided via the parameter `dirname`. This function supports both absolute and relative paths, including navigation through parent directories using `..` and self-references using `.`.

At the beginning of the function, a `std::stringstream` object named `ss` is initialised with the given `dirname`, allowing the string to be split by the delimiter `/`, which separates each directory level. The method uses a loop with `std::getline()` to iterate over each component (token) of the path.

A temporary pointer `temp` is set to point to the current working directory at the start. As the loop iterates through the tokens:

- If the token is empty or a single period (`.`), it is ignored, since it refers to the current directory.
- If the token is a double period (`..`), the method attempts to move one level up in the directory hierarchy by setting `temp` to its parent. If `temp` has no parent (i.e., it is already at the root directory), an error message is printed, and the method returns `false` to indicate failure.
- Otherwise, the function attempts to find a subdirectory matching the token using `findSubdir()`. If such a directory exists, `temp` is updated to point to it. If not, an error message is displayed, and the method again returns `false`.

If all tokens are processed successfully, and each path segment is valid, the function updates the `currentDir` to point to the final directory reached and returns `true` to indicate a successful change directory operation.

This method is central to implementing the `cd` command, as it accurately emulates the hierarchical traversal logic of a typical file system shell.

III.2.7 Method `printWorkingDirectory()`

```
void Directory::printWorkingDir(Directory* currentDir, bool isBackground) {
    pid_t pid = fork();

    if (pid < 0) {
        std::cerr << "Error! Fork failed for pwd." << std::endl;
        return;
    }

    if (pid == 0) {
        std::vector<std::string> path;
        Directory* temp = currentDir;

        while (temp != nullptr) {
            path.push_back(temp->dirname);
            temp = temp->parent;
        }

        std::cout << "/";
        for (std::vector<std::string>::reverse_iterator it = path.rbegin();
            it != path.rend(); ++it) {
            if (*it != "/") std::cout << *it << "/";
        }
        std::cout << std::endl;

        _exit(0);
    } else {
        if (!isBackground) {
            int status;
            waitpid(pid, &status, 0);
        } else {
            std::cout << "Background process pwd started with PID: "
            << pid << std::endl;
        }
    }
}
```

The `printWorkingDir()` method emulates the behaviour of the `pwd` (print working directory) command in a Unix shell. It prints the absolute path of the current working directory, optionally executing in the background if the `isBackground` flag is set to `true`.

The method begins by forking the current process using the `fork()` system call. If the return value is negative, the fork has failed, and an appropriate error message is printed. No further action is taken in this case.

If the return value is zero, the function is now executing within the child process. A temporary pointer `temp` is initialised to the current working directory, and a `std::vectorstd::string`

named `path` is used to collect the names of all parent directories, starting from the current directory and moving upwards via the `parent` pointer. This constructs the path from the leaf to the root in reverse order.

After reaching the root, the function proceeds to print the full path. It first outputs a leading slash to indicate the root. Then, it iterates over the `path` vector in reverse using a reverse iterator, printing each directory name followed by a slash, except when the directory name itself is already the root (`"/"`), to avoid redundant slashes.

Once the path has been printed, the child process terminates by calling `_exit(0)`, which was already explained above. [1]

Back in the parent process (when `pid > 0`), if the command is not intended to run in the background, the parent waits for the child to terminate by invoking `waitpid()`. This behaviour was also explained above.

This design ensures that the `pwd` command behaves correctly in both foreground and background contexts, mimicking the control flow of an actual Linux shell.

IV CommandParser Module

IV.1 CommandParser.h

```
1  #pragma once
2  #include <string>
3  #include <vector>
4
5  enum class Operator { NONE, SEQUENTIAL, AND, OR, BACKGROUND, PIPELINE };
6
7  struct CommandSegment {
8      std::string command;
9      Operator op = Operator::NONE;
10 };
11
12 std::vector<CommandSegment> parseCommands(const std::string& input);
```

At the first line, `#pragma once` is declared. This prevents the header file from being included multiple times during the compilation process. It serves as an include guard, ensuring that the compiler only processes the contents of `CommandParser.h` once, even if it is included multiple times via different paths or intermediate headers. This helps avoid redefinition errors and reduces compilation time. [3]

The file begins by including two standard headers: `<string>` and `<vector>`, which are necessary for defining the types used in the following declarations.

An `enum class` named `Operator` is then defined. It enumerates the different types of operators that can appear between commands in a shell input string. These include:

- `NONE` – No operator (a standalone command).
- `SEQUENTIAL` – Multiple commands separated by a semicolon (`;`).
- `AND` – Conditional execution if the previous command succeeds (`&&`).

- `OR` – Conditional execution if the previous command fails (`||`).
- `BACKGROUND` – Background execution (`&`).
- `PIPELINE` – Pipeline operator (`|`).

Following that, a `struct` named `CommandSegment` is declared. Each `CommandSegment` represents a single command parsed from the user's input, along with an associated operator that describes how it is connected to the following command. The default operator is set to `Operator::NONE`, indicating that no special linkage is applied unless explicitly parsed.

Finally, the function prototype `parseCommands()` is declared. This function takes a shell input string as an argument and returns a `std::vector` of `CommandSegment` objects. Each element in the vector represents a parsed command segment, preserving the original order and structure of the user's input with its corresponding operator. This parser is foundational for interpreting and executing complex shell command sequences, including conditionals, pipelines, and background tasks.

IV.2 `CommandParser.cpp`

```cpp
#include "CommandParser.h"

#include <sstream>

std::vector<CommandSegment> parseCommands(const std::string& input) {
    std::vector<CommandSegment> result;
    std::istringstream iss(input);
    std::string token, commandBuffer;

    while (iss >> token) {
        if (token == ";") {
            result.push_back({commandBuffer, Operator::SEQUENTIAL});
            commandBuffer.clear();
        } else if (token == "&&") {
            result.push_back({commandBuffer, Operator::AND});
            commandBuffer.clear();
        } else if (token == "||") {
            result.push_back({commandBuffer, Operator::OR});
            commandBuffer.clear();
        } else if (token == "&") {
            result.push_back({commandBuffer, Operator::BACKGROUND});
            commandBuffer.clear();
        } else if (token == "|") {
            result.push_back({commandBuffer, Operator::PIPELINE});
            commandBuffer.clear();
        } else {
            if (!commandBuffer.empty()) commandBuffer += " ";
            commandBuffer += token;
        }
    }

    if (!commandBuffer.empty()) {
        result.push_back({commandBuffer, Operator::NONE});
    }

```

```
36      return result;
37  }
```

IV.2.1 Standard Library

The `CommandParser.cpp` file uses the `<sstream>` headerfile from the C++ Standard Library. This header provides string stream manipulation, particularly the `std::istringstream` class used in this context. An object of this class allows the input string to be tokenised using whitespace as the split token, enabling easy parsing of shell command strings. This approach is aimed to be efficient in modern C++ for processing space-separated input sequences.

IV.2.2 Method `parseCommands()`

The `parseCommands()` function is the core parser responsible for converting a raw shell input string into a structured sequence of command segments, each accompanied by an operator. This enables the simulated shell to interpret compound commands involving sequencing, conditional execution, background tasks, and pipelines. The function takes a single `std::string` parameter named `input`, which represents the raw command entered by the user. It begins by declaring a vector of `CommandSegment` objects named `result`, which will store the parsed output. An `std::istringstream` object named `iss` is constructed using the input string, allowing the command to be tokenised by whitespace. Two local string variables are used: `token`, which holds each extracted word or symbol from the input, and `commandBuffer`, which accumulates the current command being parsed. The function enters a loop where it reads each token from the input:

- If the token is a recognised shell operator (such as `;` or `&&`), the current contents of `commandBuffer` are paired with the appropriate `Operator` enum value and pushed into the result vector. The enum value defined is at the section IV.1. The buffer is then cleared to begin parsing the next command.
- If the token is not an operator, it is appended to `commandBuffer`. To ensure proper spacing between words in the command, a space is inserted before each additional token (unless the buffer is empty).

After the loop completes, any remaining content in `commandBuffer` (representing the final command in the sequence) is pushed into the result vector with `Operator::NONE`, indicating no explicit operator follows it. In summary, `parseCommands()` processes the user's input string into a structured and machine-readable format, supporting multiple command types and logical operators. This modular structure is essential for handling complex user input and for implementing logical control flow in the shell's command execution phase.

Example: Parsing a Compound Shell Command

In this section, the aim is to show how a compound shell command is processed through the `parseCommands` function.

Consider the following input string:

```
ls -al && mkdir test || echo "mkdir failed" & cd test
```

This input contains multiple commands joined by different operators: `&&`, `||`, `&`, and an implicit `NONE` operator for the final command. After parsing with `parseCommands()`, the result is a vector of `CommandSegment` structs as follows:

```
[
    { command = "ls -al", op = Operator::AND },
    { command = "mkdir test", op = Operator::OR },
    { command = "echo \"mkdir failed\"", op = Operator::BACKGROUND },
    { command = "cd test", op = Operator::NONE }
]
```

Each segment correctly represents a portion of the original string, along with its associated control operator. This structure allows the shell to:

- Run `ls -al`, and only run `mkdir test` if the first command succeeds.
- Run `echo "mkdir failed"` if `mkdir test` fails.
- Run `echo "mkdir failed"` in the background.
- Run `cd test` independently as the final sequential command.

This example illustrates the parsing logic, showing a well-structured and accurate interpretation of the shell input.

V CommandRunner.cpp

```cpp
#include "Directory.h"
#include "MainRunner.h"
#include "CommandParser.h"
#include "CommandRunner.h"
#include "PipelineRunner.h"

#include <iostream>
#include <string>

extern Directory*   root;
extern Directory*   home;
extern Directory*   currentDir;
extern std::string  username;
extern std::string  devicename;

void MainRunner() {
    std::cout << username << "@" << devicename << ": " << currentDir->dirname << " $ ";
    std::string input;
    std::getline(std::cin, input);
    std::vector<CommandSegment> segments = parseCommands(input);

    bool lastSuccess = true;

    for (unsigned long i = 0; i < segments.size(); ++i) {
        const CommandSegment& segment = segments[i];

        if (i > 0) {
            Operator prev = segments[i - 1].op;

            if (prev == Operator::AND && !lastSuccess) continue;
            if (prev == Operator::OR && lastSuccess) continue;
        }

        if (segment.op == Operator::BACKGROUND) {
```

```
35              lastSuccess = runCommand(segment.command, true);
36              continue;
37          }
38
39          if (segment.op == Operator::PIPELINE) {
40              std::vector<std::string> pipeline;
41              pipeline.push_back(segment.command);
42
43              while (i + 1 < segments.size() && segments[i].op == Operator::PIPELINE) {
44                  ++i;
45                  pipeline.push_back(segments[i].command);
46              }
47
48              lastSuccess = runPipeline(pipeline);
49              continue;
50          }
51
52          lastSuccess = runCommand(segment.command);
53      }
54  }
```

V.1 MainRunner.cpp

This source file serves as the entry point for command execution in the shell program. It is responsible for handling user input, parsing command sequences, evaluating logical and control operators, and executing commands accordingly, either as standalone commands, background tasks, or pipelines. Although the `CommandRunner.h` headerfile is as well declared, in this report, we do not go over it since it only contains simple pre-declaration codes.

V.1.1 Header Inclusions

The line 1-8 in this file includes several headers, each serving a purpose in the program:

- `Directory.h`: Defines the directory structure and filesystem navigation logic.
- `MainRunner.h`: Declaration of the `MainRunner()` function.
- `CommandParser.h`: Provides the `parseCommands()` function and `CommandSegment` struct.
- `CommandRunner.h`: Defines the logic for executing individual commands.
- `PipelineRunner.h`: Contains the implementation of multi-command pipelines.
- `<iostream>` and `<string>`: Standard headers for input/output and string handling.

V.1.2 Global Variables

Several global variables outside this particular code file are referenced via `extern` declarations in the line 10-14:

- `Directory* root, home, currentDir` - Represent the filesystem's root, home, and current working directories.
- `std::string username, devicename` - Used for prompt customisation, displaying the current user and device name.

V.1.3 Function `runCommand()`

The `runCommand()` function orchestrates the command-line shell's main loop, handling input parsing and execution logic. The steps are as follows:

Prompt Display and Input Retrieval The first line in the function prints the shell prompt in the format:

```
username@devicename: current_directory $
```

This is followed by retrieving the entire line of input from the user using `std::getline()` and storing it in the `input` string.

Command Parsing The input string is then passed to the `parseCommands()` function, which returns a vector of `CommandSegment` objects. Each segment represents a command along with an corresponding operator (e.g., `&&`, `|`, `&`, etc.).

Execution Loop and Logical Evaluation A loop iterates through each command segment to determine whether and how it should be executed. A Boolean variable, `lastSuccess`, tracks whether the previous command succeeded. This allows logical operators like `&&` and `||` to behave correctly:

- If the previous operator is `AND` and the last command failed, the current command is skipped.
- If the previous operator is `OR` and the last command succeeded, the current command is skipped.

Background Execution If the current segment's operator is `BACKGROUND`, the command is run in the background by passing `true` to the `runCommand()` function. The loop then continues to the next iteration without waiting for the background process to complete. Note that the `true` parameter will lead to background execution, as implemented in the `runCommand()` function.

Pipeline Execution If the current segment's operator is `PIPELINE`, the function enters a nested loop to collect all consecutive pipeline-linked commands into a vector. This vector is then passed to `runPipeline()`, which handles the setup of inter-process communication. The loop index `i` is manually incremented to skip over all processed pipeline segments. The `runPipeline()` function will be explained again in the later section.

Regular Command Execution If no special operator applies, the command is executed via `runCommand()` in the foreground, and the success status is stored in `lastSuccess`.

VI `PipeRunner` Module

```
#include "PipelineRunner.h"

#include <iostream>
#include <sstream>
#include <vector>
#include <cstring>
#include <unistd.h>
#include <sys/wait.h>
```

```
std::vector<char*> parseArgs(const std::string& command) {
    std::istringstream iss(command);
    std::string token;
    std::vector<char*> args;

    while (iss >> token) {
        char* arg = new char[token.size() + 1];
        std::strcpy(arg, token.c_str());
        args.push_back(arg);
    }

    args.push_back(nullptr);
    return args;
}

void freeArgs(std::vector<char*>& args) {
    for (char* arg : args) delete[] arg;
}

bool runPipeline(const std::vector<std::string>& commands) {
    int numCmds = commands.size();
    int prevPipe[2] = { -1, -1 };

    for (int i = 0; i < numCmds; ++i) {
        int pipefd[2];
        if (i < numCmds - 1) {
            if (pipe(pipefd) < 0) {
                std::cerr << "Pipe creation failed\n";
                return false;
            }
        }

        pid_t pid = fork();
        if (pid < 0) {
            std::cerr << "Fork failed\n";
            return false;
        }

        if (pid == 0) {
            if (prevPipe[0] != -1) {
                dup2(prevPipe[0], STDIN_FILENO);
                close(prevPipe[0]);
                close(prevPipe[1]);
            }

            if (i < numCmds - 1) {
                close(pipefd[0]);
                dup2(pipefd[1], STDOUT_FILENO);
                close(pipefd[1]);
            }

            std::vector<char*> args = parseArgs(commands[i]);
            execvp(args[0], args.data());

            std::cerr << "Command not found: " << args[0] << "\n";
            freeArgs(args);
            _exit(1);
```

```
        }

        if (prevPipe[0] != -1) {
            close(prevPipe[0]);
            close(prevPipe[1]);
        }

        if (i < numCmds - 1) {
            prevPipe[0] = pipefd[0];
            prevPipe[1] = pipefd[1];
        }
    }

    for (int i = 0; i < numCmds; ++i) {
        wait(nullptr);
    }

    return true;
}
```

VI.1 `PipelineRunner.cpp`

This source file is responsible for implementing support for shell pipelines (using the pipe operator `|`). It enables multiple commands to be chained together such that the output of one command is passed as input to the next, mirroring the behaviour of a Unix shell pipeline.

VI.1.1 Header Inclusions

The following headers are included:

- `PipelineRunner.h` – Declares the `runPipeline()` function and related helpers.
- `<iostream>` – Used for printing error messages.
- `<sstream>` – Used to split command strings into arguments.
- `<vector>` – Provides the vector template used throughout the file.
- `<cstring>` – Used for `std::strcpy()` to create `char*` arguments.
- `<unistd.h>` – Provides system calls like `fork()`, `pipe()`, and `execvp()`.
- `<sys/wait.h>` – Provides the `wait()` system call to wait for child processes.

VI.1.2 Helper Function `parseArgs()`

Two helper functions are used in this code: `parseArgs()` function to parse arguments and `freeArgs()` function to free the functions that are used in the argument parsing process. The `parseArgs()` function takes a shell command as a `std::string` and tokenises it into individual arguments:

```
std::vector<char*> parseArgs(const std::string& command) {
    std::istringstream iss(command);
    std::string token;
    std::vector<char*> args;

    while (iss >> token) {
```

```
        char* arg = new char[token.size() + 1];
        std::strcpy(arg, token.c_str());
        args.push_back(arg);
    }

    args.push_back(nullptr);
    return args;
}
```

- A `std::istringstream` is used to split the command string by whitespace.
- For each token, a dynamically allocated `char*` array is created and added to a vector.
- A final `nullptr` is appended to mark the end of the arguments for use in `execvp()`.

This function is essential because `execvp()` requires arguments in the form of a `char*[]` array with a `nullptr` terminator.

VI.1.3 Helper Function `freeArgs()`

This function deallocates the memory allocated by `parseArgs()`:

```
void freeArgs(std::vector<char*>& args) {
    for (char* arg : args) delete[] arg;
}
```

It iterates through the vector and calls `delete[]` on each pointer, preventing memory leaks in the case of failed command execution.

VI.1.4 Function `runPipeline()`

The core logic of pipeline execution is handled in this function:

```
#include "PipelineRunner.h"

#include <iostream>
#include <sstream>
#include <vector>
#include <cstring>
#include <unistd.h>
#include <sys/wait.h>

std::vector<char*> parseArgs(const std::string& command) {
    std::istringstream iss(command);
    std::string token;
    std::vector<char*> args;

    while (iss >> token) {
        char* arg = new char[token.size() + 1];
        std::strcpy(arg, token.c_str());
        args.push_back(arg);
    }

    args.push_back(nullptr);
    return args;
}
```

```
void freeArgs(std::vector<char*>& args) {
    for (char* arg : args) delete[] arg;
}

bool runPipeline(const std::vector<std::string>& commands) {
    int numCmds = commands.size();
    int prevPipe[2] = { -1, -1 };

    for (int i = 0; i < numCmds; ++i) {
        int pipefd[2];
        if (i < numCmds - 1) {
            if (pipe(pipefd) < 0) {
                std::cerr << "Pipe creation failed\n";
                return false;
            }
        }

        pid_t pid = fork();
        if (pid < 0) {
            std::cerr << "Fork failed\n";
            return false;
        }

        if (pid == 0) {
            if (prevPipe[0] != -1) {
                dup2(prevPipe[0], STDIN_FILENO);
                close(prevPipe[0]);
                close(prevPipe[1]);
            }

            if (i < numCmds - 1) {
                close(pipefd[0]);
                dup2(pipefd[1], STDOUT_FILENO);
                close(pipefd[1]);
            }

            std::vector<char*> args = parseArgs(commands[i]);
            execvp(args[0], args.data());

            std::cerr << "Command not found: " << args[0] << "\n";
            freeArgs(args);
            _exit(1);
        }

        if (prevPipe[0] != -1) {
            close(prevPipe[0]);
            close(prevPipe[1]);
        }

        if (i < numCmds - 1) {
            prevPipe[0] = pipefd[0];
            prevPipe[1] = pipefd[1];
        }
    }

    for (int i = 0; i < numCmds; ++i) {
        wait(nullptr);
    }
```

```
    return true;
}
```

The function accepts a vector of command strings, each representing a stage in the pipeline (e.g., `ls | grep hello`). It returns a boolean value indicating success or failure of the process.

Pipe Setup and Fork Loop The function first initialises a pair of file descriptors, `prevPipe`, to keep track of the previous command's pipe. Then it enters a loop over each command:

- If the current command is not the last one, a new pipe is created using `pipe(pipefd)`.
- A new process is created using `fork()`.

Child Process Logic If `pid == 0`, the child process is responsible for executing one stage of the pipeline:

- If a previous pipe exists, `dup2()` is used to redirect standard input from that pipe.
- If this is not the final command, the next pipe is redirected to standard output.
- The command is parsed using `parseArgs()`, and executed with `execvp()`.
- If `execvp()` fails, an error message is printed, memory is freed, and the process exits with code 1.

Parent Process Logic In the parent process:

- Any previous pipe is closed after forking the child.
- The current pipe becomes the new `prevPipe` for the next iteration.

Process Waiting After all commands have been forked, the parent process calls `wait()` once for each command to ensure all children terminate properly before continuing.

- Output from `commands[i]` becomes input for `commands[i+1]`.
- Pipes are set up only between adjacent commands.
- Each command runs in its own process context.

This design allows for highly flexible and concurrent execution of piped shell commands.

VII Conclusion

VII.1 Aims

This code was written to simulate the standard Linux shell behaviour. It successfully replicated the behaviour of commands, trying best to mimic the actual procedures that happen inside the system for each commands.

Also, one of the main goals was to implement the commands from scratch, and avoiding the `exec()` function as much as possible. As a result, the program was successful in implementing the `Directory` class properly, and the commands deriving out of the class.

VII.2 Limitation

However, this program still does have certain limitations. For example, as mentioned above, we failed to implement various commands, since it is aimed from-scratch implementation.

The behaviour of the `mkdir` method is slightly off compared to the actual behaviour, since the current code only works in the volatile memory space, unlike the actual Linux shell. The actual `mkdir` method creates a child process, which is different from the current implementation.

VII.3 Considerations

The extended implementation of the commands and the kernel behaviours may be implemented in the future.

VII.4 Scope of the use of LLMs

The following parts of this project was written fully, or partially by the LLM.

- `Makefile` – Fully written by LLM.
- Partial syntactical revisions or code completion for C++ code.
- Grammatical correction in this LaTeX based document – Used the *Writefull* AI, a plug-in in the Overleaf platform.

No other parts of the code or the report, such as the overall logic, ideas, and the report contents were written by the LLM.

VIII Additionals: `Makefile` and the `run.sh` script

The scripts will not explained thoroughly, but is provided. The `Makefile` also contains the C++ versions and compiler environment.

VIII.1 `Makefile`

```
CXX = g++
CXXFLAGS = -std=c++17 -Wall -Wextra

SRCDIR = .
BUILDDIR = build

SRCS = $(wildcard $(SRCDIR)/*.cpp)

OBJS = $(patsubst $(SRCDIR)/%.cpp, $(BUILDDIR)/%.o, $(SRCS))

TARGET = $(BUILDDIR)/main

all: $(TARGET)

$(TARGET): $(OBJS)
        @mkdir -p $(BUILDDIR)
        $(CXX) $(CXXFLAGS) -o $@ $^

$(BUILDDIR)/%.o: $(SRCDIR)/%.cpp
        @mkdir -p $(BUILDDIR)
        $(CXX) $(CXXFLAGS) -c $< -o $@
```

```
run: $(TARGET)
	./$(TARGET)

clean:
	rm -rf $(BUILDDIR)
```

VIII.2 `run.sh` script

```
set -e

make clean
make
make run
```

References

[1] IBM C/C++ Runtime Library Reference link

[2] C++ in Visual Studio Language Reference link

[3] C++ Reference link