CoE 164

Computing Platforms

05b: Rust Paths and Files

FILES

A **file** is a collection of data treated as a single entity saved on some memory storage.

In Linux, "everything"* is a file. This includes hardware connected to the computer and data streams like internet connections.

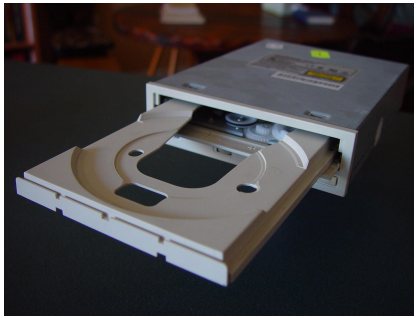
* If it is not a file, it is otherwise a process.

LINUX FILE TYPES

Regular

Text and data files

Computer programs



Special

Hardware

Inter-process communication sockets and pipes

Directory

Container of files

LINUX FHS

All Linux-based operating systems follow the **Filesystem Hierarchy Standard (FHS)**. This organizes the different files that make up the operating system into logically consistent groups.

The *root* directory / encloses all of the files and folders.

Example

```
/
  |- bin
  |- boot
  |- dev
  |- etc
  |- home
  |- lib
  |- media
  |- mnt
  |- opt
  |- proc
  |- usr
  |   |- bin
  |   |- sbin
  |- var
  |- ...
```

PATH

A **path** is a string that uniquely identifies a location in some directory structure.

There are different ways of representing a path depending on the operating system (OS). Most common of these are the Linux- and Windows-style paths.

Linux:
```
/usr/bin/g++
```

Windows:
```
C:\Users\Admin\scoop\apps\msys2\current\ucrt64.exe
```

Example

PATH: RELATIVITY

OSes have a *root* directory which encloses all of the files and folders in a system. Paths that are written containing the root at its leftmost is called an **absolute path**. In contrast, a **relative path** is written with a directory *not* the root at its leftmost.

Linux:

```
/usr/bin/g++ // Absolute
./Downloads // Relative
```

Windows:

```
C:\Users\Admin\scoop\apps\msys2\current // Absolute
.\%USERPROFILE%\Downloads // Relative
```

Example

PATH: CREATE

The `Path` struct represents a path. The module automatically represents the path specifically to the OS it is run from.

Path manipulation is part of the standard library.

```
use std::path::Path;

let current_dir = Path::new("/");
let some_path = Path::new("/home/admin/Downloads");

println!("Current Directory: {}", current_dir.display());
```

Example

PATH: MANIPULATE

`Path` has methods that allow creation of new paths. This is similar to running the `cd` command in a terminal.

```
let download_path = Path::new("/home/admin/Downloads");

// /home/admin
let parent_path = download_path.parent().unwrap();

// /home/admin/Downloads/Programs
let child_path = download_path.join("Programs");
```

Example

PATH: MANIPULATE

Some methods of `Path` return a `PathBuf` struct that enables in-place editing. The relationship between `Path` and `PathBuf` deref-wise is the same as with `&str` and `String`.

```
let download_path = Path::new("./src");

let child_path = download_path.join("assets"); // ./src/assets
child_path.pop(); // ./src
child_path.push("my_lib"); // ./src/my_lib

let child_abs_path = child_path.canonicalize().ok();
```

Example

PATH: MANIPULATE

`PathBuf` can also be created from scratch. This is useful when the path will be built in multiple parts of the program.

```
use std::path::PathBuf;

let download_path = PathBuf::new();

// /home/admin
download_path.push("/home");
download_path.push("/admin");
```

Example

PATH: ERRORS

Most methods of `Path` and `PathBuf` return a `Result` enum. The most common error encountered is nonexistence of a path.

```
let download_path = Path::new("./src");

let dp_abs_o = download_path.canonicalize().ok(); // Absolute path
let dp_is_exist_o = download_path.try_exists().ok();
```

Example

FILE: OPEN

A file can be opened by creating a `File` struct from a path. The path can either be a `String` or a `Path`.

```
use std::fs::File;
use std::path::Path;

let fh_path = Path::new("/home/admin/Downloads/README.txt");
let mut fh = File::open(fh_path).ok();
```

Example

FILE: OPEN OPTIONS

Files are opened for reading only by default. An `OpenOptions` struct can be created instead to set the different read or write options.

```
use std::fs::{File, OpenOptions};
use std::path::Path;

let fh_path = Path::new("/home/admin/Downloads/README.txt");
let mut fh = OpenOptions::new()
    .create(true).write(true).truncate(true)
    .open(&fh_path)
    .ok();
```

Example

FILE: READ

`File` has methods that allow reading of files treated as a readable string or collection of bytes. This usually requires allocation of another variable where the contents will be stored.

```
use std::io::{Read};

let mut fh = File::open("hello.txt")?;
let mut fh_buf = String::new();
fh.read_to_string(&mut fh_buf)?;

println!("-----Contents-----\n{}", fh_buf);
```

Example

FILE: LINE READ

A common method of reading files is reading by line. For memory efficiency, a `BufReader` should be used.

```
use std::io::{BufRead, BufReader};

let mut fh = File::open("hello.txt")?;
let fh_lines = BufReader::new(fh).lines();

for each_line in fh_lines.flatten() {
    println!("{}", each_line);
}
```

Example

FILE: FILE POINTER

The underlying structure of a `File` is a *file pointer*, which is a one way pointer to a specific byte location of the file. This can be indirectly manipulated from its `Seek` trait.

```
use std::io::{Seek};

let mut fh = File::open("hello.txt")?;
let mut fh_buf: Vec <u8> = vec![];
let mut fh_str = String::new();
fh.read_to_end(&mut fh_buf); // Read as bytes
fh.rewind(); // Move fp to beginning again
fh.read_to_string(&mut fh_str); // Read as string
```

Example

FILE: CREATE

Similar to reading a file, a file can be created using the `File` or `OpenOptions` structs.

```
use std::io::{Write};

let mut fh = File::create("hello.txt")?;
let mut fh2 = OpenOptions::new()
    .create(true).write(true).truncate(true)
    .open("hello.txt")?;
```

Example

FILE: WRITE

`File` has methods that allow writing of files treated as a collection of bytes. To write a formatted string, use the `write!` macro.

```
use std::io::{Write};

let mut fh = File::create("hello.txt")?;
fh.write_all(b"Hello world!");
write!(fh, "3 + 3 = {}", 3 + 3);
```

Example

RESOURCES

- The Rust Book
- Rust by Example

CoE 164

Computing Platforms

05b: Rust Paths and Files