Introduction

rmnShell is a basic Linux shell written in C, designed to handle three internal and five external commands.

The program consists of the main shell.c file, along with 5 external c files (and corresponding header files), one for each external command. Further, there are two structures (`splitStruc` and `flagStruc`), designed for obtaining the outputs generated by flags.c and tokeniser.c. Finally, the makefile complies the program and executes it. The shell operates by running the `shell()` function, which is the heart of the entire program. References are commented in the code whenever required. Linux man pages, stack exchange, and tutorialspoint were general sources for a lot of information in general.

File-by-file description

shell.c

The `shell()` function operates through a while loop centred around the volatile variable `active`. The shell is active by default, but ceases to be so if a system interrupt is called in a non cat command context (more on that later).

Each command is processed the same way: `fgets()` retrieves the input, which is tokenised (using `strtok`) and stored in the character pointer array `splitString[512][512]`. The newline character is removed from the end of the first word in the input command (which is useful for processing commands like `ls`), and then flag detection takes place.

The first word of the input is then compared with command names to detect which command has been entered. For internal commands, the function is directly executed from there, while for external commands, the process is more complicated (detailed later). I will first be describing the internal commands:

Internal commands

- cd()
    - Flags:
        - `-L`: Force symbolic links to be followed (default)
        - `-P`: use physical directory structure
    - Corner cases & Bugs:
        - Both flags entered (default followed)
        - Same flag entered twice
        - Invalid flags entered
        - Flag case independence
        - Blank input after cd (redirects to `the /home/$(getenv("USER"))` like in bash cd)
        - Invalid directory input
        - Other errors using `perror` (via `errno.h`)

    - Functioning:
        Uses `chdir()` to change to requested directory
- `pwd()`
    - Flags:
        - `-L`: Print symbolic directory structure
        - `-P`: Print physically resolved directory structure (default)
    - Corner cases & Bugs:
        - Both flags entered (default followed)
        - Flag case independence
        - Same flag entered twice
        - Invalid flags entered
        - Other errors using `perror` (via `errno.h`)
    - Functioning:
        Uses `getcwd()` to change to requested directory

- `echo()`
    - Flags:
        - `-N`: Ignore ending newline
        - `-H`: Prints help
    - Corner cases & Bugs:
        - Fork failures are caught (`pid_t<0`)
        - Both flags entered (default followed)
        - Same flag entered twice
        - Invalid flags entered
        - Space added after every iterative word is printed, except the last one (which would cause a minor visual bug)
        - Input is `<echo >` gives a new line, but `<echo -n>` and `<echo>` don't
    - Functioning:
        Prints the user input received

External Commands:

Follow the same format in general:

- Call `fork()` and then `execl()` the file by generating the name and location of its `*.out` file, which are of the same format (`name_.out`).
- The parent, detected by `pid>0`, waits.
- For threading, a `syscaller` function, which generates the path of the function and is called by `pthread_create()`. `Pthread_join()` suspends the parent till the called function has executed meanwhile.
- Corner cases & Bugs:

- Invalid commands are handled (applies to all commands, internal and external) with an error messages
  Fork failures are caught (`pid_t<0`)
- If current working directory changes, the value of `getcwd()` would change too, and the generated path for the function in `syscaller` would lead to a non existent file. An `originalLoc` string generated when the program starts prevents this
- `System()` calls would fail and try to run my given command (say, `rm`) with the `t` if `<rm&t>` was entered into bash, giving an invalid command (`t`) error. `&` is replaced with `t` to combat this. Also looks funny (*`cattt`*).
- For `cat`, I wanted to implement inputting text into a new file the same way as the bash command. For example:

  ```
  [bashShell]$ Cat > newfile
  This text was
  Entered into the file
  Entry can be ended by using the system interrupt
  (Ctrl+C)
  ^C
  [bashShell]$
  ```

  For this, system interrupts are handled using the interrupter function, which sets `active` to `0` if `cat` is not the currently operating command (i.e. if the sysinterrupt is called during normal operation), but merely catches the interrupt and continues the shell if the interrupt was called during cat processing. Used `signal(SIGINT, interrupter)` for that.

External Command Files (`cat, date, ls, mkdir, rm`)

- If the main function is called by `execl`, the input string is placed in `argv[0]` directly. It gets tokenised and added to `splitString[512][512]` in the `splitStruc` tokens
- If the thread is called by the corresponding `syscaller` in shell.c, the arguments are iteratively added to `splitString[512][512]` in the `splitStruc` tokens
- Flags are detected, and the function for the command is called.
- If the thread flag (`&t`) is detected, `pthread_exit()` is called

- **cat_.c**
  - Flags:
    - `-N`: Line numbering
    - `> $(filename)`: newfile
  - Corner cases & Bugs:

        - Invalid flags entered
        - Both flags entered
        - Empty input after cat
        - No filename after >
        - Signal interrupt detection
        - Accidental ending newline (caused by the newline the user needs to create before triggering the sysinyterrupt)
        - Input is `<echo >` gives a new line, but `<echo -n>` and `<echo>` don't
    - Functioning:

      Prints contents of a file, or creates a new file. Uses `fopen()` and `fprintf()` for file operations

- **date_.c**
    - Flags:
        - `-U`: UTC-Time
        - `-R`: RFCs 5322 date
    - Corner cases & Bugs:
        - Invalid flags entered
        - Both flags entered
        - arguments given to date beyond flags
    - Functioning:

      Loads current time into a `time_t` object and converts it to various formats, using `ctime(), mktime(), and strftime()`.
- **ls_.c**
    - Flags:
        - `-a`: Don't ignore files with names starting from '. '
        - `-R`: Recursively list cwd as well as all of its subdirectories
    - Corner cases & Bugs:
        - Both flags entered
        - Invalid flags entered
        - Arguments given after specifying a directory

          `<ls -flag1 -flag2 directory extra arguments which are ignored>`
        - Symlinks present: Error message
        - `nftw()` handles changes to files made during read automatically using `errno.h`
        - Other errors handled by `perror`
    - Functioning:
        - Loads requited directory to a `DIR*` object using `opendir()`, and reads its contents using `readdir()`, which are stored in a `dirent*` structure. The names of the files are displayed by accessing the structure's `d_name` attribute.

        - For recursively accessing subdirectories, a lister function is passed into `nftw()`, where it's executed in every node during `nftw`'s file tree walk.
- **ls_.c**
    - Flags:
        - `-M`: Set mode manually (default is A, i.e. `rwx`)
        - `-V`: Verbose (print name of each created directory)
    - Corner cases & Bugs:
        - Both flags entered
        - Entered mode is longer than three characters
        - Mode is improperly entered/is invalid
        - No dirname entered (missing operand)
        - Invalid flags entered
        - Multiple directories entered (are supported, i.e. get created as with the bash command)
        - One of multiple directory creation operations fails (error message shown and program moves to making next directory name inputted)
        - Other errors handled my `perror`
    - Functioning:
        - Creates directories using `mkdir()`, with modes for `R, W, and X` being defined earlier. Runs a for loop through the command in the case of multiple dirname entry to create them one by one.

- **rm_.c**
    - Flags:
        - `-D`: Directory (remove empty directory)
        - `-R`: Recursive (remove non empty directory)
    - Corner cases & Bugs:
        - Both flags entered (conflicting flag entry)
        - If `-D` is entered but directory is non-empty, or both flags are entered, a confirmation message is shown before deleting file
        - Invalid flags entered
        - Multiple directories entered (are supported, i.e. get created as with the bash command)
        - One of multiple directory creation operations fails (error message shown and program moves to making removing next filename inputted)
        - `nftw()` handles changes to files made during read automatically using `errno.h`
        - Other errors handled my `perror`
    - Functioning:
        - Used `remove()` to delete files

        - For recursively accessing subdirectories, a remover function is passed into `nftw()`, where it's executed in every node during `nftw`'s file tree walk.

- **flags.c**
    - Uses `strcmp` to detect flags, including > for `cat`
    - Returns indices of flags and `flag1Taken` and `flag2Taken` integers via a `flagStruc` output.

- **tokeniser.c**
    - Uses `strtok` to split a char array by spaces.
    - Returns a character pointer array and the length of array using a `splitStruc` output.

Test Cases

```
pwd -L garbage //show that garbage values don't affect functioning
+ functioning of L flag

mkdir -M -V rwx dir1 //Improper mode input

mkdir -v -m rwx dir1 //case insensitive flags, flag functionality

cd -P dir1 //flag functionality: -P

cat > file1 //file creation

this is a test

file please

^C //Control+C, Enter

cat&t -N file1 //thread + -N flag functionality

mkdir&t lorom ipsum guys //make multiple directories + threads

ls -a //flag -a

cd .. //unique cd input

rm -d guys //flag

ls

cd ..

pwd //non flag output

rm -d dir1 //confirmation + flag
```

```
n
ls&t -R //threads + flag
ls dir1 //targeted ls
rm -r dir1 //flag
cd //empty input
pwd //check cwd's functionality
date&t -r //flag + thread
e //exit
```