I have implemented a shell using C libraries and Linux system calls like fork(), execv(), waitpid() etc.

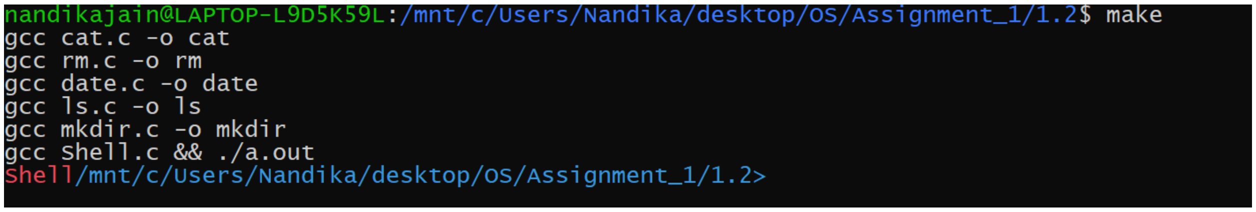
Compiling the code through Makefile

In order to run the file through Makefile, a default flag is provided, which produces an executable and runs the binary.

make

Make command creates executable for the programs for the external commands and links it with the main code to produce a.out. Running the a.out runs the shell.

(Note: In order to clear files like *./a.out history.txt cat rm date ls mkdir,* clear flag is provided, (make clear))



```
nandikajain@LAPTOP-L9D5K59L:/mnt/c/Users/Nandika/desktop/OS/Assignment_1/1.2$ make
gcc cat.c -o cat
gcc rm.c -o rm
gcc date.c -o date
gcc ls.c -o ls
gcc mkdir.c -o mkdir
gcc shell.c && ./a.out
Shell/mnt/c/Users/Nandika/desktop/OS/Assignment_1/1.2>
```

Snippet of the command which is used to run the shell and how the shell interface looks like (Note- The given Shell handles blank inputs, i.e if nothing was entered, it skips through the command and doesn't throw an error if a command was used that the shell doesn't support).

Internal Commands

Internal commands are those which are interpreted by the shell program itself without requiring a different program to handle the expected operations.

Internal commands implemented are as follows-

cd

Description

cd is used to change the current working directory to a specified folder.

Flags implemented

1. -P
   This flag uses physical directory structure without following symbolic links.
2. -e
   When this flag is used, if the -P option is supplied, and the current working directory cannot be determined successfully, exit with a non-zero status

Handling errors and corner cases

perror and ***errno*** are used to handle the errors for the system call.

1. While using chdir, *ENOENT* is handled, that is that it handles the error when the file to navigate to doesn't exit.

2. *EACESS* is handled, i.e. while navigating, if one of the directories' search permission is denied.
3. It handles the error if too many arguments are passed by the user.

Usage
To use relative or absolute paths, the command is used as it is with the flags required as cd [-P [-e]] *path*. You can also go directly to the shell home using the command as cd [-P [-e]]. The path can also be relative to shell home using the ~ prefix.

Syscalls used
1. getcwd()
2. chdir()

Assumptions made
1. It makes the assumption that the user doesn't enter cd - to go to the last directory instead uses cd ..

echo

Description
Displays a line of text on the terminal.

Flags implemented
1. -n
   This flag doesn't append a new line.
2. -e
   This flag enables interpretation of backslash escapes.

Handling errors and corner cases
perror is used to handle the errors for the system call.
1. *EINTR* is handled, which handles the error if the call was interrupted before any data was written.
2. It handles the cases where both the flags are implemented at once, i.e. if both -n and -e are used.
3. It handles the cases for any no of arguments in the input.

Usage
The command can be called as echo [-e] [-n] *text*

Syscalls used
1. write() (Since printf uses write system call)

Assumptions made
1. It makes the assumption that if a string starts at ", it should also end at ".

2. It makes the assumption that the flags should be specified right after the command echo.

history

Description

Maintains a history of all the commands used across all the sessions of the shell.

Flags implemented

1. -c
   This flag clears the history list by deleting all of the entries.
2. -d *OFFSET*
   This flag deletes the history entry at position *OFFSET*. Negative offsets count back from the end of the history list.

Handling errors and corner cases

perror is used to handle the errors for the system call

1. It handles the error *EBADF*, which occurs if fd is a non-valid file identifier which is used to read/write/close/open a file(history.txt)
2. It handles *EFAULT*, when the buffer used is outside the accessible memory space.
3. It handles the error for inputting more arguments than required.
4. It handles if negative value is given to print history.
5. It handles if while using -d flag, the user entered offset which is more than the total commands in the history.
6. It handles the case where in the user cd's into some other directory and access history(stored the path of history file in a variable HISTORY_PATH)

Usage

To view the entire history, use command history. To view the last OFFSET commands, use history OFFSET. To delete the command at OFFSET, use history -d OFFSET. To clear the entire history, use history -c.

Syscalls used

1. open()
2. close()
3. read()
4. write()

Assumptions made

1. It makes the assumption that if -c flag is called, it will not only delete the history for that session but will also delete history across all the sessions.
2. It makes the assumption that the offset entered by the user for -d flag will be positive.

pwd

Description

This command gives the present working directory of the shell.

Flags implemented

1. -L
   This flag uses PWD from the environment, even if it contains symlinks.
2. -P
   This flag avoids all symlinks.

Handling errors and corner cases
Perror and ***errno*** is used to handle the errors for the system call.

1. It handles ENOMEM, which occurs when user memory cannot be mapped.
2. It handles the error of ERANGE, which occurs when there is not enough space for storing the path.
3. It handles the error if the user enters more no of arguments than required.

Usage
To use this command, simply run it as `pwd [-P] [-L]`

Syscalls used

1. getcwd()
2. getenv()

(Note- No assumptions are being made for pwd)

exit

Description
This command exits the user from the shell.

Usage
Use as `exit`

Syscalls used

1. exit()

(Note- No assumptions are being made for exit)

External Commands

External commands are commands that aren’t implemented by the shell itself. The shell looks for their implementation and runs them as new child process
The following syscalls have been used in all the external commands to run them

1. fork()
2. waitpid()
3. execv()

In order to handle the external commands, a child process is created every time any of the external commands are used on the shell. It creates a child process using fork system call and using execv system call which is used to replace the current process with a new process, we pass the arguments to the external commands program and access them. After execution exit system call is used which ends the child process, meanwhile using waitpid, the parent program waits for the execution of the child program to end.
Error handling is done for waitpid(), fork() and execv(). These system calls return a negative value on error, and it is handled using ***errno*** and ***perror***.

date

Description
Using this command, the user can print the system date and time.

Flags implemented

1. -u
   Prints the coordinated universal time (UTC)
2. -R
   outputs date and time in RFC 5322 format

Handling errors and corner cases
Perror and ***errno*** is used to handle the errors for the system call.

1. It handles the error *EFAULT* which occurs when time points outside the available memory space.
2. It handles the error where time() and localtime() system calls failed to return valid values.
3. It handles the case when more arguments are inputted than the required.
4. It handles the position of the flags, i.e the flags can be used at anywhere after the command date.
5. It handles the case if more than one flag is used.

Usage
The command can be called as date [-u] [-R].

Syscalls used

1. time()
2. localtime()
3. memset()

(Note- No assumptions are being made for date)

cat

Description
This command concatenates file and prints on the standard output

Flags implemented

1. -E
   This flag displays $ at end of each line.
2. -n
   This flag numbers all output lines.

Handling errors and corner cases

Perror and ***errno*** is used to handle the errors for the system call.

1. It handles the error *EBADF*, which occurs if fd is a non-valid file identifier which is used to open the file mentioned in the arguments which is to be read and written on the terminal or if a file is not open for reading.
2. It handles the error of EFAULT, wherein the path that is mentioned is outside the accessible memory space.
3. It handles the error if no file is given to be opened.
4. It handles opening multiple files on the terminal.
5. It handles multiple flags together.
6. It handles the position of flags, i.e. the flags can be used anywhere after the command cat.
7. It handles relative paths to open a file.

Usage

The command can be called as `cat [-E] [-n] file1 file2 …` to concatenate files in those order

Syscalls used

1. open()
2. close()
3. read()

Assumptions made

1. It doesn’t give the option of reading the standard input when no file is mentioned, instead it asks the user to type in another command.

<u>ls</u>

Description

This command is used to list directory contents.

Flags implemented

1. -a
   Using this flag, the command does not ignore entries starting with .
2. -A
   Using this flag, the command does not list implied . and ..

Handling errors and corner cases

Perror and ***errno*** is used to handle the errors for the system call.

1. It handles the error of *ENOENT,* which occurs if no directory or file exists with the path name.
2. It handles the error of *ENOTDIR*, which occurs when the given path doesn't correspond to a directory.
3. It handles listing contents from different paths.
4. It handles the position of flags, i.e. the flags can be used anywhere after the command rm.

Usage

The command can be called as ls [-a] [-A] path1 path2 ... to list the contents of files or directory in that order.

Syscalls used

1. opendir()
2. readdir()

Assumptions made

1. It makes the assumption that even if one path is given, it will print the path asked by the user and then the contents with -a and -A flag.

rm

Description

This command can be used to remove files or directories.

Flags implemented

1. -i
   This flag prompts before every removal
2. -v
   This flag explains what is being done.

Handling errors and corner cases

Perror and ***errno*** is used to handle the errors for the system call.

1. It handles the error of *ENOENET*, which occurs when no file with that pathname exists.
2. It handles the error of *EROFS*, which occurs when pathname refers to a read-only file system
3. It handles the case where no file is given to be deleted.
4. It handles removing multiple files on the terminal.
5. It handles multiple flags together.

6. It handles the position of flags, i.e. the flags can be used anywhere after the command rm.
7. It checks if the given specified argument is a file and not a directory.

Usage
The command can be called as `rm [-i] [-v] file1 file2 ...`

Syscalls used
1. remove()

(Note- No assumptions are being made for rm)

mkdir

Description
Using this command, we can make directories.

Flags implemented
1. -p
   This flag creates all the directories recursively
2. -v
   This flag prints a message for each created directory

Handling errors and corner cases
Perror and ***errno*** is used to handle the errors for the system call.
1. It handles the error of *EEXIST*, which occurs when the pathname already exists.
2. It handles the error of *ENOENT*, which occurs when the pathname does not exist.
3. It handles the case where no directory is given to be created.
4. It handles making multiple directories on the terminal.
5. It handles multiple flags together.
6. It handles the position of flags, i.e. the flags can be used anywhere after the command mkdir.

Usage
The command can be called as `mkdir [-p] [-v] file1 file2 ...`

Syscalls used
1. getcwd()
2. mkdir()
3. chdir()

Assumptions made
1. It makes the assumption that while using -p flag, if ***mkdir -p a/b/c*** is used, then one cannot use the file name ‘a’ as the starting while using the -p flag.