Assignment 1

CS330: Operating Systems

1 Execute This!!!

1.1 Execute a Command (15 Marks)

In this part, you will be exploring how to execute a command passed as an argument. You need to complete the implementation of the following function in `Part1/1.1/executeCommand.c`

SYNOPSIS

```
int executeCommand(char* cmd);
```

DESCRIPTION

This function should create a child process to execute the command `cmd`. The command `cmd` will only contain a single program name followed by its options. A single space will separate options in `cmd`. (e.g., "ls -a -l"). The function must get the path to the executable in `cmd` from the environment variable `CS330_PATH`. This environment variable is a colon-delimited string of paths in which to look for the executable.

Eg. `"/usr/bin:/usr/local/bin"`

RETURN VALUE

On success, return the status code of `cmd`. On failure, return -1.

ERRORS

Check for possible errors such as lack of permission, no binary file found, etc. Print "UNABLE TO EXECUTE" and exit in case of any error.

SYSTEM CALLS AND LIBRARY FUNCTIONS

You **must only use** the below mentioned APIs to implement this part of the assignment. Refer to man page of these APIs to know about their usage.

- fork
- execv
- wait
- exit
- stat
- pipe
- open
- close
- read
- dup2
- WEXITSTATUS
- strtok
- getenv
- strcpy
- strcat
- strcmp
- strlen
- printf
- fgets
- malloc

TESTING

Run the script `Part1/1.1/runTests.sh` for running the provided sample test cases. You can see output of your program for `TEST CASE N` in the file `outputN.txt`.

```
$ ./runTests.sh
```

Make sure they are all passing before submission.

1.2 Execute In Parallel (25 Marks)

Since you now know how to execute a command, you are required to execute multiple commands in parallel in this part. You need to complete the implementation of the following function in `Part1/1.2/executeInParallel.c`.

SYNOPSIS

```
int executeInParallel(char *infile, char *outfile);
```

DESCRIPTION

The function takes paths to an input file (`infile`) and an output file (`outfile`) as arguments. The input file will contain at most 50 commands to be executed, one per line. The function must create child processes, one for each command, which will execute the commands. The child processes must execute in parallel. The parent should not wait for one child to finish executing before forking another child.

The child processes must communicate the outputs to the parent process. The parent process must write these outputs to the `outfile` in the order of the commands specified in `infile`.

The full path to executables for the commands must be found out using `CS330_PATH` environment variable as in Part 1.1.

RETURN VALUE

On success, return 0. On failure, return -1.

ERRORS

If a command was not found in the path or if it failed to execute due to some errors write "UNABLE TO EXECUTE" as its output to the `outfile`.

SYSTEM CALLS AND LIBRARY FUNCTIONS

You **must only use** the below mentioned APIs to implement this part of the assignment. Refer to man page of these APIs to know about their usage.

- fork
- execv
- wait
- stat
- pipe
- open
- close
- read
- dup2
- strtok
- getenv
- strcpy
- strcat
- strcmp
- fgets
- malloc
- printf

TESTING

Run the script `Part1/1.2/runTests.sh` for running the provided sample test cases. You can view the output of your program for each of the test cases in `Part1/1.2/outputs` directory. Make sure they are all passing before submission.

2 Lets Design a Game, Rock Paper Scissors!!!

Game Rules

Two players secretly choose either `paper`, `scissors` or `rock`. They then reveal their choice. An umpire decides who wins as follows:

- `Paper beats rock`
- `Rock beats scissors`
- `Scissors beat paper`
- `Matching choices draw`

The winner gets a point or no points to both players in case of a draw.

2.1 Umpire program (25 Marks)

In this part of the assignment, you must complete the umpire program in `Part2/2.1/umpire.c`.

The program should take two player executable names as arguments and decide the winner based on the game rules.

SYNOPSIS

```
$ ./umpire player1 player2
```

Details on Player:

You are given two player programs as part of the assignment, `player1` and `player2`. These are executables compiled from the source file `Part2/2.1/player.c`. Run **make** inside `Part2/2.1/` to compile the player programs.

We will use the following representations for the possible throws:
`0` → `rock`, `1` → `paper`, `2` → `scissors`.

Player processes when given 'G','O', ' \0' (3 bytes) as `stdin` will output a move (1 byte character with value either `'0'`, `'1'` or `'2'`) to `stdout`. Players will exit on `EOF`.

DESCRIPTION

The umpire program will do the following:

- Create a child process for each player. The child process will `exec` the specified player program (once the stdin and stdout are set up appropriately to communicate with the umpire)
- Play a rock paper scissors match for **ten** rounds. In each round the umpire will write "GO" to the stdin of the players. Then the umpire will read the moves from the stdout of each player. The umpire will keep score for both players.
- After ten rounds, print the score of the first player followed by a space and then the score of the second player.

OUTPUT

```
6 4
```

ERROR

Check for possible errors like the player executable not being found, or the pipe creation or exec call failing. If there were no errors, exit returning 0. In case of errors, umpire should exit by returning -1.

SYSTEM CALLS AND LIBRARY FUNCTIONS

You **must only use** the below mentioned APIs to implement this part of the assignment. Refer to man page of these APIs to know about their usage.

- pipe
- fork
- dup2
- close
- execv
- execl
- write
- open
- exit
- read
- printf

TESTING

Run the script `Part2/2.1/runTests.sh` for running the provided sample test cases. Make sure they are all passing before submission.

2.2 Tournament (35 Marks)

Create a modified version of `umpire` in `Part2/2.2/umpire2.c` that can conduct a rock-paper-scissors tournament.

SYNOPSIS

```
$ ./umpire2 [-r rounds_per_match] players.txt
```

DESCRIPTION

The -r option (if it exists) is an integer N, the number of rounds to be played per match. If -r does not appear, default to N=10.

The first line in `players.txt` specifies the number of players, P, participating in the tournament. The next P lines are the names of the player executables. The player list may contain duplicates. The maximum length of a player program name will be 100 characters. There must be at least two competitors specified. We give each competitor an ID starting from 0 to P-1 in the order shown in players.txt.

Here is a possible `players.txt` file for a tournament with 3 competitors:

```
players.txt
-----------
3
player1
player2
player2
```

Your program will do the following:

- Print the IDs of all the competitors (0 to P-1) in the tournament.
- Create one child process per competitor and setup communication with the umpire using pipes like before.
- In each level of the tournament, pair up the competitors and referee matches between them in parallel. The competitors that haven't lost any matches are termed "`active.`"

  The following algorithm pairs the participants:

  Order the players in increasing player ID. Iteratively pair the lowest ID active and unpaired competitor with the second-lowest ID active and unpaired competitor. In case of an odd number of active competitors, one player will be given walkover i.e. that player will be allowed to move to next level of the tournament without playing against any player in current level. Ignore the competitor who got walkover during pairing.

  To decide which player will be given the walkover, call the provided function in `Part2/2.2/gameUtils.c`

  ```
  getWalkOver(int numActivePlayers);
  /* Returns a number between [1, numActivePlayers] */
  ```

  Example: if $p0$, $p4$, $p6$ are the active players, and if getWalkOver(3) returns 2, that means the 2nd active process, $p4$, gets the walkover. $p0$ and $p6$ are paired.
- For each level, the umpire will oversee multiple matches in parallel. E.g., If there are two pairs $(p0, p1)$, $(p2, p3)$, then the umpire will communicate with $(p0, p1)$ to referee their first round, and then it will communicate with $(p2, p3)$ to referee their first round. While there are rounds still left to play in the match, the umpire will alternate refereeing rounds $(p0, p1)$ and $(p2, p3)$ until the matches are over.
- The player with the smaller ID is declared as the winner in case of a TIE.
- Terminate any losing competitors at each level of the tournament.
- Print the competitor IDs, in increasing order, that get to advance to the next level in a space separated line. Include the competitor that got a walkover.

ERROR

In case of any errors like a player is not able to participate i.e. `exec` fails, tournament will not take place and umpire should exit by returning -1. Exit returning 0 on success.

SYSTEM CALLS AND LIBRARY FUNCTIONS

You **must only use** the below mentioned APIs to implement this part of the assignment. Refer to man page of these APIs to know about their usage.

- fork
- execv
- dup2
- pipe
- open
- close
- read
- write
- wait
- exit
- fcntl
- printf
- malloc
- free
- atoi

TESTING

Run the script `Part2/2.2/runTests.sh` for running the provided sample test cases. Make sure they are all passing before submission.

Example: Tournament with 5 players

`Level 1 of the tournament`
$p0$, $p1$, $p2$, $p3$, $p4$ will be paired as $(p0, p1)$, $(p2, p3)$. Let $p4$ be the player that got a walkover.. This is because getWalkOver(5) returned 5, which means the 5th **active** player ($p4$) gets the walkover.
Let the winners be $p0$, and $p2$. These 3 players (including $p4$) will play in the next level of the tournament and remaining players will be terminated.

`Level 2 of the tournament:`
Among $p0$, $p2$, $p4$, one player will get walkover. Assume getWalkOver(3) returns 2 which means the 2nd active player gets the walkover i.e. $p2$ gets the walkover. So the following pairing will take place $(p0, p4)$. Let $p0$ win the match. $p4$ is terminated. $p0, p2$ will move on to the finals.

`Level 3 of the tournament:`
$p0$, $p2$ are paired. Let the scores be tied. Since $p0$ is the lower ID process. Therefore, $p0$ is declared the winner.

```
OUTPUT
------
p0 p1 p2 p3 p4
p0 p2 p4
p0 p2
p0
```

3 Deliverables

- Part 1.1 : `executeCommand.c`
- Part 1.2 : `executeInParallel.c`
- Part 2.1 : `umpire.c`
- Part 2.2 : `umpire2.c`

Put these files in a directory with your roll number as the name. Zip this directory and upload.