The Islamic university of Gaza

Faculty of engineering

ECOM 4010

OS LAB

Project 1

Name:

Mai Yaser EL_Lahham

STUD. NO.:

220161814

Submitted to:

Eng. Nisreen Maher Abo_Shammalah

Command method:

```
static int command(int input, int first, int last)
{
int pipettes[2];

/* Invoke pipe */
pipe( pipettes );
pid = fork();

/SCHEME:STDIN --> O --> O --> O --> STDOUT/
if (pid == 0) {
        if (first == 1 && last == 0 && input == 0) {
                // First command
                dup2( pipettes[WRITE], STDOUT_FILENO );    // put its output on the pipe(wite on the pipe).
        } else if (first == 0 && last == 0 && input != 0) {
                // Middle command
                dup2(input, STDIN_FILENO);  //take its input from the first command output(read from the pipe).
                dup2(pipettes[WRITE], STDOUT_FILENO);      // put its output on the pipe(write on the pipe).
        } else {
                // Last command.
                dup2( input, STDIN_FILENO ); //take its input from the second comand output(read from the pipe).
        }

        if (execvp( args[0], args) == -1)             //if the command isn't defined.
                _exit(EXIT_FAILURE);            // If child fails.
}
if (input != 0)
        close(input);// close the zero file because we do a dup2 above(we don't need 0 file any more)

// Nothing more needs to be written
close(pipettes[WRITE]); // we finished the write.

// If it's the last command, nothing more needs to be read
if (last == 1)
        close(pipettes[READ]); //if the last = 1 it means no more inputs so close the read.

return pipettes[READ];              //return the output.
}
```

- The command method is for the pipe state, it takes the output of the argument, and then it writes it on the pipe, and then the second arg. Can read the output of the first one from the pipe.
- it retunes the output of the arg. To the main method.

Run method:

```
static int run(char* cmd, int input, int first, int last)
{
split(cmd);                    //its a method defind below.
if (args[0] != NULL) {
        if (strcmp(args[0], "quit") == 0) //quit.
                exit(0);
        if (strcmp(args[0], "pause") == 0)
                getchar();
        n += 1;
        return command(input, first, last); // do the pipe method above.
}
return 0;
}
```

- The run method takes the argument saved in 'args' array from the split method, checks the special conditions args. Like pause and quit here, counts the number of the args in the command to clean it later, and finally pass it to the command to run it.

SkipWhite method:

```
static char* skipwhite(char* s) // skip the space .
{
while (isspace(*s)) ++s;
return s;
}
```

- If it's a space make the pointer point on the next char

Split method:

```
static void split(char* cmd)
{
cmd = skipwhite(cmd);
char* next = strchr(cmd, ' ');                  //pointer on the next arg.
int i = 0;

while(next != NULL) {
        next[0] = '\0';                         // put a /0 instead of | to define it as a string.
        args[i] = cmd;                          //store the next arg. in args[i].
        ++i;
        cmd = skipwhite(next + 1);              //skip the write space.
        next = strchr(cmd, ' ');                //find the next word(arg).
}

if (cmd[0] != '\0') {
        args[i] = cmd;
        next = strchr(cmd, '\n');
        next[0] = '\0';
        ++i;
}

args[i] = NULL;
}
```

- The split takes the command read from the user and take the first arg. and store it in the ‘args’ array .

Clean method:

```
static void cleanup(int n)
{
int i;
for (i = 0; i < n; ++i)
        wait(NULL); //it waits to terminate the file.
}
```

- clean the files after executing the args.

Main method:

```
int main()
{
printf("SIMPLE SHELL: Type 'quit' or send EOF to exit.\n");
while (1) {
        /* Print the command prompt */
        printf("$> ");
        fflush(NULL);

        /* Read a command line */
        if (!fgets(line, 1024, stdin))                  //to read from the user
                return 0;

        int input = 0;
        int first = 1;

        char* cmd = line;
        char* next = strchr(cmd, '|');                  /* Find first '|' */

        while (next != NULL) {
                /* 'next' points to '|' */
                *next = '\0';
                input = run(cmd, input, first, 0);

                cmd = next + 1;
                next = strchr(cmd, '|');                /* Find next '|' */
                first = 0;
        }
        input = run(cmd, input, first, 1);              // the last =1.
        cleanup(n);
        n = 0;
}
return 0;
}
```

- It reads the command from the user put it in ‘cmd’ variable , pass it to ‘run’ method, keep reading and executing args. until it ends.

The whole project:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* The array below will hold the arguments: args[0] is the command. */
static char* args[512];
pid_t pid;
int command_pipe[2];
#define READ  0
#define WRITE 1

/*
  Handle commands separatly
  input: return value from previous command (useful for pipe file descriptor)
  first: 1 if first command in pipe-sequence (no input from previous pipe)
  last: 1 if last command in pipe-sequence (no input from previous pipe)

  EXAMPLE: If you type "ls | grep shell | wc" in your shell:
     fd1 = command(0, 1, 0), with args[0] = "ls"
     fd2 = command(fd1, 0, 0), with args[0] = "grep" and args[1] = "shell"
     fd3 = command(fd2, 0, 1), with args[0] = "wc"

  So if 'command' returns a file descriptor, the next 'command' has this
  descriptor as its 'input'.
 */
static int command(int input, int first, int last)
{
```

```c
 */
static int command(int input, int first, int last)
{
        int pipettes[2];

        /* Invoke pipe */
        pipe( pipettes );
        pid = fork();

        /SCHEME:STDIN --> 0 --> 0 --> 0 --> STDOUT/
        if (pid == 0) {
                if (first == 1 && last == 0 && input == 0) {
                        // First command
                        dup2( pipettes[WRITE], STDOUT_FILENO ); // put its output on the pipe(wite on the pipe).
                } else if (first == 0 && last == 0 && input != 0) {
                        // Middle command
                        dup2(input, STDIN_FILENO); //take its input from the first command output(read from the pipe).
                        dup2(pipettes[WRITE], STDOUT_FILENO); // put its output on the pipe(write on the pipe).
                } else {
                        // Last command.
                        dup2( input, STDIN_FILENO ); //take its input from the second comand output(read from the pipe).
                }

                if (execvp( args[0], args) == -1) //if the command isn't defined.
                        _exit(EXIT_FAILURE); // If child fails.
        }
        if (input != 0)
                close(input);// close the zero file because we do a dup2 above(we don't need 0 file any more)
```

```
                if (execvp( args[0], args) == -1) //if the command isn't defined.
                        _exit(EXIT_FAILURE); // If child fails.
        }
        if (input != 0)
                close(input);// close the zero file because we do a dup2 above(we don't need 0 file any more)

        // Nothing more needs to be written
        close(pipettes[WRITE]); // we finished the write.

        // If it's the last command, nothing more needs to be read
        if (last == 1)
                close(pipettes[READ]); //if the last = 1 it means no more inputs so close the read.

        return pipettes[READ];//
}
```

```
/* Final cleanup, 'wait' for processes to terminate.
   n : Number of times 'command' was invoked.*/
static void cleanup(int n)
{
        int i;
        for (i = 0; i < n; ++i)
                wait(NULL); //it waits to terminate the file.
}

static int run(char* cmd, int input, int first, int last);
static char line[1024];
static int n = 0; /* number of calls to 'command' */
```

```
int main()
{
        printf("SIMPLE SHELL: Type 'quit' or send EOF to exit.\n");
        while (1) {
                /* Print the command prompt */
                printf("$> ");
                fflush(NULL);

                /* Read a command line */
                if (!fgets(line, 1024, stdin)) //to read from the user
                        return 0;

                int input = 0;
                int first = 1;

                char* cmd = line;
                char* next = strchr(cmd, '|'); /* Find first '|' */

                while (next != NULL) {
                        /* 'next' points to '|' */
                        *next = '\0';
                        input = run(cmd, input, first, 0);

                        cmd = next + 1;
                        next = strchr(cmd, '|'); /* Find next '|' */
                        first = 0;
                }
                input = run(cmd, input, first, 1); // the last =1.
                cleanup(n);
                n = 0;
        }
```

```
                input = run(cmd, input, first, 1); // the last =1.
                cleanup(n);
                n = 0;
        }
        return 0;
}

static void split(char* cmd);

static int run(char* cmd, int input, int first, int last)
{
        split(cmd); //its a method defind below.
        if (args[0] != NULL) {
                if (strcmp(args[0], "quit") == 0) //quit.
                        exit(0);
                if (strcmp(args[0], "pause") == 0) //quit.
                        getchar();
                n += 1;
                return command(input, first, last); // do the pipe method above.
        }
        return 0;
}

static char* skipwhite(char* s) // skip the space .
{
        while (isspace(*s)) ++s;
        return s;
}
```

```
static void split(char* cmd)
{
        cmd = skipwhite(cmd);
        char* next = strchr(cmd, ' '); //pointer on the next arg.
        int i = 0;

        while(next != NULL) {
                next[0] = '\0'; // put a /0 insted of | to defin it as a string.
                args[i] = cmd; //store the next arg in args[i].
                ++i;
                cmd = skipwhite(next + 1); //skip the wite space.
                next = strchr(cmd, ' '); //find the next word(arg).
        }

        if (cmd[0] != '\0') {
                args[i] = cmd;
                next = strchr(cmd, '\n');
                next[0] = '\0';
                ++i;
        }

        args[i] = NULL;
}
```