Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

Sistemas Operativos

INFORME PROYECTO #1: MINISHELL

Estudiantes:

César Carios

Jhonatan Homsany

Estructura del código fuente y funcionalidades más relevantes

El código fuente consta de 5 funciones: 2 para ejecutar programas, 1 de lectura de comandos, 1 de lectura de entrada y la función main.

```
> int executeCommand(char *command, char *argumentArray[]){···

> int executeProgram(char *arguments[], int size){···

> void readLine(char input[], pid_t mainProcess){ ···

> void readUserInput(){···
> int main(){···
```

- *ExecuteProgram y ExecuteCommand:* El objetivo de esta función es generalizar la ejecución de los comandos introducidos en el minishell. Cuando el proceso padre llama a esta función crea un proceso hijo que será el encargado de hacer la llamada a otra función que llama a *execvp*. El objetivo de esta función "puente" es enviar los argumentos para *execvp* de acuerdo con la cantidad de argumentos leídos en la entrada; para ello, esta función también recibe como parámetro el tamaño de la entrada para el programa que se quiere ejecutar.

```
int executeCommand(char *command, char *argumentArray[]){

    return execvp(command, argumentArray);
}
```

```c
int executeProgram(char *arguments[], int size){
    pid_t child = fork();
    if(child == 0){
        if(size == 1){
            return executeCommand(arguments[0], NULL);
        }
        else{
            return executeCommand(arguments[0], arguments);
        }
    }
    else{
        wait(NULL);
    }
}
```

- *ReadUserInput:* Tal como lo indica su nombre, esta función es la encargada de tomar el input del usuario. Esto lo hace el proceso padre mediante una estructura de control while, que seguirá pidiendo el input del usuario hasta que el mismo teclee “salir”. Esta función guarda el ID del proceso principal de todo el programa, es decir, el padre de todos los demás procesos que se crean a lo largo de la ejecución del minishell. En este punto del código se llama a la función clave de este programa: *readLine*, que se explicará a continuación.

```c
void readUserInput(){
    pid_t mainProcess = getpid();
    char input[200];
    while(1){
        fgets(input, sizeof(input), stdin);
        if(strcmp(input, "salir\n") != 0){
            readLine(input, mainProcess);
        }
        else{
            printf("Finishing...\n");
            return;
        }
    }
}
```

- *ReadLine:* Inicialmente esta función guarda en un arreglo cada uno de los comandos que se han introducido en la entrada por el usuario. Los comandos y los operadores (si hay) los guarda en posiciones diferentes del arreglo. Luego se recorre el arreglo en busca de algún operador para establecer si se ejecutarán uno o dos programas, además se guarda la posición donde está el operador. Considerando la posición del operador, se separa la entrada en dos arreglos diferentes para separar sus argumentos y comando de programa. En el caso de que únicamente se tenga que ejecutar un programa, un puntero de arreglo auxiliar nos indica la cantidad de argumentos que se deben guardar.

  Tras guardar las entradas de los programas en los arreglos, llega el momento de ejecutar los operadores PIPE (|), OR (||) y AND (&&). Para poder distinguir entre operadores, se emplea la variable que almacenó la posición de estos y se usó la función *strcmp* para verificar a cuál de ellos corresponde. A continuación, se detalla la implementación de estos operadores:

  a. OR (||): El operador OR llama inicialmente a la función executeProgram y verifica su valor de retorno. Si esta función retorna –1 significa que la ejecución del comando falló. En cuyo caso, el proceso hijo que se creó en esa función se encarga de ejecutar nuevamente a la función executeProgram para el segundo programa. Nuevamente, se verifica si hubo algún error y se muestra por pantalla. Por último, se termina la ejecución de los procesos hijos que podrían seguir vivos en ese punto del programa.

  b. AND (&&): Para el operador AND se presentaron complicaciones tales como el paso de mensajes entre los procesos hijos y el proceso padre cuando sucedía algún error con el primer programa. Pues, no se podía realizar una validación similar a la del operador OR. Por el funcionamiento de execvp, cuando se realiza un cambio de contexto entre el proceso hijo y el programa, el proceso hijo muere cuando se termina el programa. Para solucionar esto, se implementó una tubería en la cual se escribe el caracter "!" sí y sólo sí el proceso hijo que ejecutó el primer programa sigue activo tras finalizar la ejecución del primer programa enviado. De esta manera, si en la tubería no se lee dicho caracter, es porque no sucedió ningún error y

se debe proceder con la ejecución del segundo programa; la validación de errores del segundo programa si se realizó como en el operador OR.

c. PIPE (|): Para este operador, inicialmente se establece una tubería en la cual los procesos estarán escribiendo o leyendo las salidas de los programas que se ejecuten según corresponda.
Luego, se crea un proceso hijo el cual manipula las tuberías y duplica el descriptor de archivos de escritura. Al hacer esto, garantizamos que su salida sea escrita en la tubería y no por consola. Seguidamente, el segundo proceso duplica el buffer de lectura para poder recibir esa entrada al momento de ejecutarse.

   Mientras lo anterior sucede, el proceso padre se encuentra esperando a que cada uno de los hijos termine su ejecución para verificar los errores.

   Para la validación de errores, si alguno de los procesos hijos seguían existiendo en el programa tras ejecutar execvp, son terminados por medio de exit retornando un valor de –1 para indicar que hay un error. Cuando el padre termina de esperar por los hijos, guarda sus códigos de retorno en unas variables de estado y muestra los errores correspondientes.

Fork y Excec en el código

El minishell se apoya en estas dos llamadas al sistema. Usamos fork para crear procesos hijos, estos procesos hijos son los encargados de ejecutar los comandos que se reciben por la entrada del usuario, luego de hacerlo mueren y el padre sigue ejecutando el while explicado al principio de este documento. Por otro lado, el excecvp ejecuta un programa especificado por un arreglo que contiene el comando con sus argumentos realizando un cambio de contexto.

Casos de Prueba

1.Comando make, make clean y ls. Compilaremos el programa en el propio programa.

```
ccarios@LAPTOP-OK5E59NR:~/Sistemas Operativos Linux/MiniShell$ ./Proyecto
make
gcc -c Proyecto.c -o Proyecto.o
Proyecto.c: In function ‘executeProgram’:
Proyecto.c:26:9: warning: implicit declaration of function ‘wait’ [-Wimplicit-function-declaration]
   26 |         wait(NULL);
      |         ^~~~
Proyecto.c: In function ‘readLine’:
Proyecto.c:86:17: warning: argument 2 null where non-null expected [-Wnonnull]
   86 |                 execvp(argumentsFirstProgram[0], NULL);
      |                 ^~~~~~
In file included from Proyecto.c:8:
/usr/include/unistd.h:599:12: note: in a call to function ‘execvp’ declared ‘nonnull’
  599 | extern int execvp (const char *__file, char *const __argv[])
      |            ^~~~~~
Proyecto.c:98:13: warning: implicit declaration of function ‘waitpid’ [-Wimplicit-function-declaration]
   98 |             waitpid(firstChild, &statusFirstChild, 0);
      |             ^~~~~~~
Proyecto.c:109:25: warning: argument 2 null where non-null expected [-Wnonnull]
  109 |                         execvp(argumentsSecondProgram[0], NULL);
      |                         ^~~~~~
In file included from Proyecto.c:8:
/usr/include/unistd.h:599:12: note: in a call to function ‘execvp’ declared ‘nonnull’
  599 | extern int execvp (const char *__file, char *const __argv[])
      |            ^~~~~~
gcc -o Proyecto Proyecto.o
```

Haremos ls para ver si se creó Proyecto.o

```
ls
Makefile  Proyecto  Proyecto.c  Proyecto.o  README.md  test.txt
```

Ejecutaremos make clean para remover el .o y lo listaremos para verificar que lo borró.

```
make clean
rm -rf Proyecto.o
ls
Makefile  Proyecto  Proyecto.c  README.md  test.txt
```

2.Comando cat, grep, sort, mkdir, rmdir, date, pwd , sudo y operadores.

Veremos el contenido de un archivo de texto con cat, lo ordenaremos alfabéticamente y luego aplicaremos algunos operadores.

```
cat test.txt
Zorro
Abeja
Mono
Cabra
Dinosaurio
```

```
sort test.txt
Abeja
Cabra
Dinosaurio
Mono
Zorro
```

```
grep Zorro test.txt && pwd
Zorro
/home/ccarios/Sistemas Operativos Linux/MiniShell
```

```
cat test.txt | grep Mono
Mono
```

```
mkdir Folder1 || date
ls -l
total 44
drwxr-xr-x 2 ccarios ccarios  4096 Jun 18 23:08 Folder1
-rw-r--r-- 1 ccarios ccarios   215 Jun 18 17:46 Makefile
-rwxr-xr-x 1 ccarios ccarios 16880 Jun 18 22:55 Proyecto
-rw-r--r-- 1 ccarios ccarios  5782 Jun 18 21:39 Proyecto.c
-rw-r--r-- 1 ccarios ccarios   895 Jun 18 18:02 README.md
-rw-r--r-- 1 ccarios ccarios    37 Jun 18 22:51 test.txt
```

```
rmdir Folder1 && date
Tue Jun 18 23:09:42 -04 2024
ls
Makefile  Proyecto  Proyecto.c  README.md  test.txt
```

3.Prueba de errores

```
abcd && echo Hola
ERROR: Invalid command.
```

```
abcd || ls -l -a ../
ERROR: Invalid command.
total 28
drwxr-xr-x 7 ccarios ccarios 4096 Jun  7 21:29  .
drwxr-x--- 6 ccarios ccarios 4096 Jun 15 22:56  ..
drwxr-xr-x 3 ccarios ccarios 4096 May 10 20:13 'Laboratorio 1 Enunciado'
drwxr-xr-x 2 ccarios ccarios 4096 May 12 20:44 'Laboratorio 1 Soluciones'
drwxr-xr-x 2 ccarios ccarios 4096 Jun  6 20:59 'Laboratorio 2 Enunciado'
drwxr-xr-x 2 ccarios ccarios 4096 Jun  7 21:33 'Laboratorio 2 Soluciones'
drwxr-xr-x 3 ccarios ccarios 4096 Jun 18 23:09  MiniShell
```

```
aaa | grep Dinosaurio
ERROR: Invalid command.
```

```
sudo apt-get upgrade
[sudo] password for ccarios:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Calculating upgrade... Done
The following packages have been kept back:
  python3-update-manager ubuntu-advantage-tools ubuntu-pro-client-l10n update-manager-core
0 upgraded, 0 newly installed, 0 to remove and 4 not upgraded.
```