PONTIFICIA UNIVERSIDAD CATÓLICA MADRE Y MAESTRA

FACULTAD DE CIENCIAS E INGENIERÍA

Sistemas Operativos II

ISC ISC365-101

L5 – Un intérprete de mandatos

Presentado por:

Junior Hernandez 2018-0999 10135069

Entregado a:

Juan Ramón Felipe Núñez Pérez

Fecha de realización: 2 de diciembre del 2022

Fecha de entrega: 3 de diciembre del 2022

Santiago de los caballeros; República Dominicana.

Introducción

Un intérprete de ordenes o de comandos, es un programa que tiene la capacidad de traducir las ordenes que introducen los usuarios, mediante un conjunto de instrucciones facilitadas por el mismo directamente al núcleo y al conjunto de herramientas que forman el sistema operativo.

Por medio de esta tarea, se implementará en c, un minishell para ser utilizado en Linux. Un Shell es un programa interactivo que interpreta comandos para usuarios para el sistema operativo, en este caso Linux. Dentro del mismo se es capaz de utilizar algunos comandos tales como, cd, dir, ren, copy, del, y exit para terminar el programa.

Marco teorico

Un intérprete de comandos, o CLI, es un programa que los usuarios de computadoras usan para ejecutar comandos de texto. Cada sistema operativo viene con su propia CLI. Por ejemplo, Windows 10 ofrece dos CLI: Símbolo del sistema y PowerShell (para usuarios avanzados). Los usuarios de Linux pueden usar el Shell para ejecutar comandos de texto (ThinkTecno, 2021).

¿Cómo funciona?

Los usuarios ingresan comandos a través del teclado. Luego, el intérprete de la línea de comandos convierte los comandos en funciones o llamadas al sistema. El sistema operativo recibe y ejecuta las respectivas llamadas. La CLI es como un traductor. El programa básicamente traduce sus instrucciones en funciones que su sistema operativo puede comprender (ThinkTecno, 2021).

Dichas instrucciones regularmente suelen crear procesos aparte del proceso que corre las CLI, por ello es necesario saber esto al momento de implementarlo en un lenguaje de programación.

Implementación.

En este caso, se debe tener en cuenta que se utiliza la función execxx() de la librería “unistd.h” para poder llamar a los comandos. La familia de funciones execxx() reemplaza la imagen del proceso actual con una nueva imagen del proceso. Por ello, al momento de utilizar exec, es necesario crear un proceso (hijo) que lo haga por el proceso principal (padre). Sus definiciones son las siguientes:

- int execl(const char *path, const char *arg, ...);
- int execlp(const char *file, const char *arg, ...);
- int execle(const char *path, const char *arg, ..., char * const envp[]);
- int execv(const char *path, char *const argv[]);
- int execvp(const char *file, char *const argv[]);
- int execvpe(const char *file, char *const argv[],
- char *const envp[]);

Explicación.

La constante tipo puntero a carácter “arg” y las variables subsiguientes en las funciones execl(), execlp() y execle() pueden considerarse arg0, arg1, ..., argn. Juntos describen una lista de uno o más punteros a cadenas terminadas en nulo que representan la lista de argumentos disponible para el programa ejecutado. El primer argumento, por convención, debe apuntar al nombre de archivo asociado con el archivo que se está ejecutando. La lista de argumentos debe terminar con un puntero NULL y, dado que se trata de funciones variadas, este puntero se debe convertir (char *) NULL (Die.Net, 2007).

Las funciones execv(), execvp() y execvpe () proporcionan una matriz de punteros a cadenas terminadas en nulo que representan la lista de argumentos disponible para el nuevo programa. El primer argumento, por convención, debe apuntar al nombre de archivo asociado con el archivo que se está ejecutando. La matriz de punteros debe terminar con un puntero NULL (Die.Net, 2007).

Las funciones execle() y execvpe() permiten a la persona que llama especificar el entorno del programa ejecutado mediante el argumento envp. El argumento envp es una matriz de punteros a cadenas terminadas en nulo y debe terminar con un puntero NULL. Las otras funciones toman el entorno para la nueva imagen de proceso de la variable externa entorno en el proceso de llamada (Die.Net, 2007).

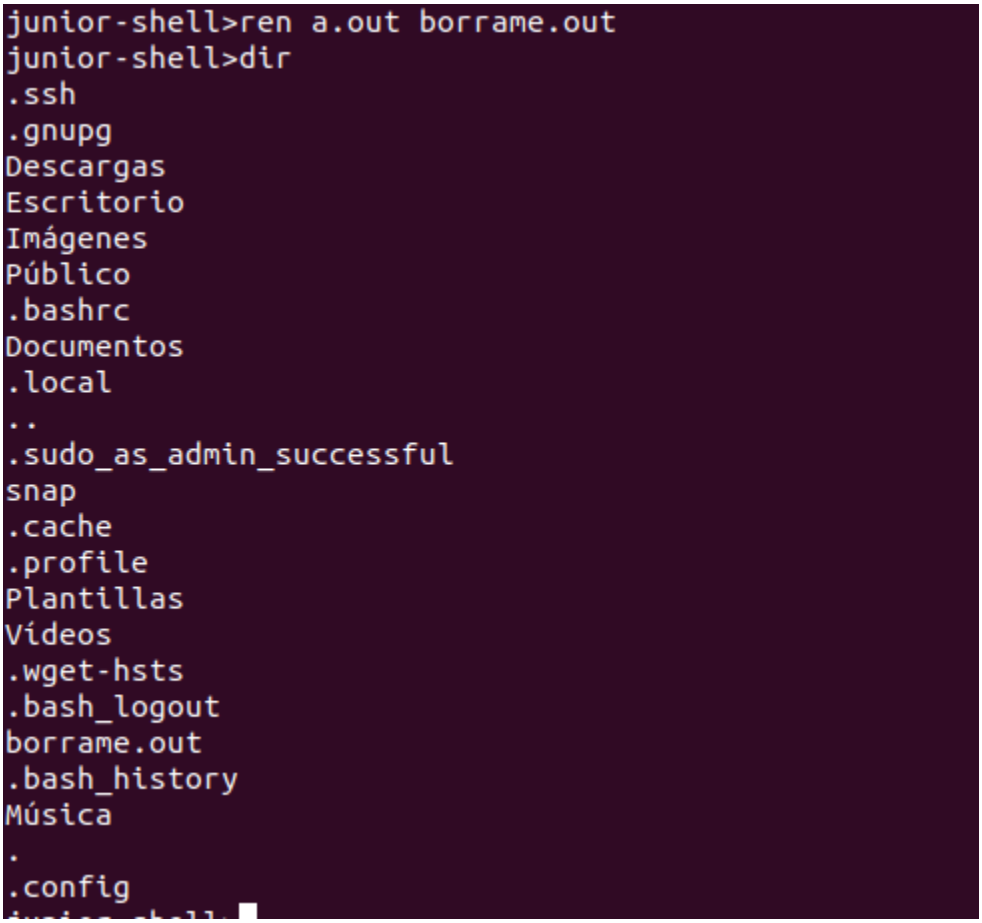
En esta práctica se hizo mayor uso de execvp() debido a que permite proporcionar una lista de argumentos a los comandos llamados. Así también como se hace uso de la función fork() para crear un proceso que ejecute esta función (Die.Net, 2007).

Desarrollo

Dentro del minishell, se puede usar comandos como cd para cambiar de directorio. También puede usar dir para ver el contenido de una carpeta, si no se especifica una carpeta específica, lo hará en la carpeta donde se encuentra el programa.

```
junior@junior-VirtualBox:~/Descargas$ ./shell
junior-shell>dir
fork_exec.c
lsdir.c
malware.py
captu.sh
execpl
lsdir
..
execpl2
fork_exec1.c
execpl3
Cronometro.sh
a.out
este.c
shell
shell.c
freq.sh
fork_exec2.c
fork_exec3.c
.
minishell.c
junior-shell>cd ..
junior-shell>dir
.ssh
.gnupg
Descargas
Escritorio
```

Así mismo tambien se puede utilizar el comando ren, para renombrar archivos



Con el comando del, eliminamos el archivo renombrado.

```
junior-shell>del borrame.out
junior-shell>dir
.ssh
.gnupg
Descargas
Escritorio
Imágenes
Público
.bashrc
Documentos
.local
..
.sudo_as_admin_successful
snap
.cache
.profile
Plantillas
Vídeos
.wget-hsts
.bash_logout
.bash_history
Música
.
.config
```

Para acabarlo, digitamos exit, y se cierra el programa.

```
junior-shell>exit
junior@junior-VirtualBox:~/Descargas$
```

Conclusion

La implementación de Minishell es una experiencia compleja y confusa, e incluso si ya conoce C, se debe considerar una explicación de lo que puede hacer el comando. Sin embargo, lo que este programa puede hacer es suficiente para llamarlo "mini-shell", porque no emula ningún comando, pero no ejecuta varios comandos que ya están disponibles en Linux. .

Codigo

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <dirent.h>
extern int errno;
#define CommandNumber 6
#define MaxCommandLine 128
#define MaxParamNumber 2
char* promptStr="junior-shell>";
char* delim=" \n";
char* helpStr= "Syntax: exit\nSyntax: cd directory\nSyntax: dir [directory]\nSyntax: del file\nSyntax: ren srcfile dstfile\nSyntax: copy srcfile dstfile\n";

enum CommandType
{
	ExitType, CDType, DirType, DelType, RenType, CopyType
};

char* commandStr[CommandNumber]= {"exit", "cd", "dir", "del", "ren", "copy"};
void exitShell(char* params[], int paramNumber);
void changeDir(char* params[], int paramNumber);
void listDir(char* params[], int paramNumber);
void delFile(char* params[], int paramNumber);
void renFile(char* params[], int paramNumber) ;
void copyFile(char* params[], int paramNumber);

void (*commandArray[CommandNumber])(char* params[], int paramNumber)=
{
	exitShell, changeDir, listDir, delFile, renFile, copyFile
};
int parseCommand(char* cmdStr, char* params[], int* paramNumber);
void dodir(char* path);
void printPrompt();
void errhandle(char* msg);
```

```
int main(int argc, char* argv[])
{
	char buf[MaxCommandLine];
	int n, paramNumber;
	int commandType;
	char* params[4];
	printPrompt();
	while ((n=read(STDIN_FILENO, buf, MaxCommandLine))>0)
	{
		buf[n]='\0';
		commandType=parseCommand(buf, params, &paramNumber);
		if (commandType==-1)
		{
			printf("Comando invalido\n%s");
		}
		else
		{
			commandArray[commandType](params, paramNumber);
		}
		printPrompt();
	}
	return 0;
}

int parseCommand(char* buf, char* params[], int* paramNumber)
{
	int i;
	*paramNumber=0;
	if ((params[*paramNumber]=strtok(buf, delim))!=NULL)
	{
		for (i=CommandNumber-1; i>=0; i--)
		{
			if (strcmp(params[*paramNumber], commandStr[i])==0)
			{
				break;
			}
		}
		if (i==-1)
		{
			return i;
		}
	}
	else
	{
		return -1;
	}
	(*paramNumber)++;
	while (1)
```

```
	{
		if ((params[*paramNumber]=strtok(NULL, delim))==NULL)
		{
			break;
		}
		(*paramNumber)++;
		if (*paramNumber==4)
		{
			return -1;
		}
	}
	return i;
}

void exitShell(char* params[], int paramNumber)
{
	exit(0);
}

void renFile(char* params[], int paramNumber)
{
	if (paramNumber!=3)
	{
		printf(helpStr);
	}
	else
	{
		if (strcmp(params[1], params[2])==0)
		{
			printf("El archivo origen %s y el archivo destino %s no pueden ser el mismo.\n",
			params[1], params[2]);
		}
		else
		{
			if (access(params[1], F_OK)<0)
			{
				printf("El archivo origen %s no existe.\n", params[1]);
			}
			else
			{
				if (rename(params[1], params[2])<0)
				{
					printf("No se pudo cambiar el nombre del archivo %s a %s\n",
					params[1], params[2]);
				}
			}
		}
	}
```

```
}

void copyFile(char* params[], int paramNumber)
{
	int status;
	if (paramNumber!=3)
	{
		printf(helpStr);
	}
	else
	{
		if (fork()==0)
		{
			execv("/bin/cp", params);
			exit(0);
		}
		else
		{
			if (wait(&status)<0)
			{
				printf("Error de espera\n");
			}
		}
	}
}

void delFile(char* params[], int paramNumber)
{
	if (paramNumber!=2)
	{
		printf(helpStr);
	}
	else
	{
		if (access(params[1], F_OK)<0)
		{
			printf("El archivo %s no existe\n", params[1]);
		}
		else
		{
			if (unlink(params[1])<0)
			{
				printf("No fue posible eliminar el archivo %s\n", params[1]);
			}
		}
	}
}
```

```
void changeDir(char* params[], int paramNumber)
{
	if (paramNumber!=2)
	{
		printf(helpStr);
	}
	else
	{
		if (chdir(params[1])<0)
		{
			if (errno==ENOTDIR||errno==ENOENT)
			{
				printf("El directorio %s no existe\n", params[1]);
			}
		}
	}
}

void dodir(char* path)
{
	DIR* dp;
	struct dirent* dirnode;
	if ((dp=opendir(path))!=NULL)
	{
		while ((dirnode=readdir(dp))!=NULL)
		{
			printf("%s\n", dirnode->d_name);
		}
	}
	else
	{
		printf("El directorio %s no existe\n", path);
	}
}


void listDir(char* params[], int paramNumber)
{
	if (paramNumber!=1&&paramNumber!=2)
	{
		printf(helpStr);
	}
	else
	{
		if (paramNumber==1)
		{
			dodir(".");
		}
```

```
                else
                {
                        dodir(params[1]);
                }
        }
}

void printPrompt()
{
        if (write(STDOUT_FILENO, promptStr, strlen(promptStr))!=strlen(promptStr))
        {
                errhandle("no pudo escribir a stdout");
        }
}

void errhandle(char* msg)
{
        perror(msg);
        exit(1);
}
```

Bibliografía

Die.Net. (2007, Agosto 5). execvp(3) - Linux man page. Obtenido de die.net:

https://linux.die.net/man/3/execvp

ThinkTecno. (2021, Enero 18). ¿Qué es un intérprete de línea de comandos? Obtenido de ThinkTecno:

https://thinktecno.com/que-es-un-interprete-de-linea-de-comandos/

Wikipedia. (2003, Junio 11). Intérprete de comandos. Obtenido de Wikipedia:

https://es.wikipedia.org/wiki/Int%C3%A9rprete_de_comandos