ESCOM-IPN

Proyecto Final
Minibash

Sistemas Operativos

Ivan Aldavera Gallaga
Laura Andrea Morales López
Erick Francisco Vázquez Nuñez

Noviembre 2019

Índice

1. Objetivo

Diseñar y desarrollar, en lenguaje C y en un sistema operativo basado en UNIX, un programa que funcione como un interprete de comandos (minishell). En la realización de éste programa se verán reflejados conocimientos de comunicación entre procesos y de llamadas al sistema como fork, exec, pipe, dup.

2. Introducción

Shell script es un intérprete de comandos de UNIX, provee una interfaz entre el usuario y el kernel para que se ocupen funciones del sistema.

El objetivo de cualquier intérprete de comandos es ejecutar los programas que el usuario teclea en el prompt del mismo. El prompt es una indicación que muestra el intérprete para anunciar que espera una orden del usuario. Cuando el usuario escribe una orden, el intérprete ejecuta dicha orden.

Bash (Bourne again shell) es un programa informático, cuya función consiste en interpretar órdenes, y un lenguaje de consola. Es una shell de Unix compatible con POSIX y el intérprete de comandos por defecto en la mayoría de las distribuciones GNU/Linux, además de macOS. También se ha llevado a otros sistemas como Windows y Android.

Su nombre es un acrónimo de Bourne-again shell (shell Bourne otra vez) haciendo un juego de palabras (born again significa nacido de nuevo) sobre la Bourne shell (sh), que fue uno de los primeros intérpretes importantes de Unix.

Hacia 1987, Bourne era el intérprete distribuido con la versión del sistema operativo Unix Versión 7. Stephen Bourne, por entonces investigador de los Laboratorios Bell, escribió la versión original. Brian Fox escribió Bash para el proyecto GNU en 1987 como sustituto libre de Bourne.112 y en 1990, Chet Ramey se convirtió en su principal desarrollador.

3. Desarrollo

Bash realiza tres principales tareas

- Inicializar: en este paso, un shell típico leería y ejecutaría sus archivos de configuración. Estos cambian aspectos del comportamiento del shell.
- Interpretar: a continuación, el shell lee los comandos de stdin (que podría ser interactivo o un archivo) y los ejecuta.

- Terminar: después de ejecutar sus comandos, el shell ejecuta los comandos de apagado, libera memoria y finaliza.

La manera en que procesaremos los comandos ingresados en nuestro minibash es de la siguiente manera.

- Imprimir el directorio actual.
- Obtienes la cadena
- Dividimos la cadena de entrada en comandos
- Comprobamos si existen tuberías, direccionamiento a archivos, etc.
- Si hay tuberías hay que manejarlas.
- Ejecutamos los comandos del sistema llamando a execvp.

4. Resultados

5. Conclusiones

En este proyecto el manejo de tuberías es la parte más complicada del mismo, el uso de execvp es el que nos permitió realizar la funcionalidad de los comandos solicitados, y cortar la cadena de entrada nos permite analizar comando por comando en la entrada, para los archivos se realizo un manejo de entrada y salida de los comandos.

Generar este proyecto nos permitió ver como podemos usar funciones para llamar al sistema y manipular la visualización de la terminal para que tenga la apariencia que deseas.

Anexos

```c
/*============================================================*/
/*=================== INSTITUTO POLITÉCNICO NACIONAL =========*/
/*=================== ESCUELA SUPERIOR DE CÓMPUTO ============*/
/*============================================================*/
/*=================== Ivan Aldavera Gallaga ==================*/
/*=================== Erick Francisco Vázquez Nuñez ==========*/
/*=================== Laura Andrea Morales López ============*/
/*============================================================*/

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

```c
#include <readline/readline.h>
#include <readline/history.h>
#include <fcntl.h>

#define MAXCOM 1000                                                    // Número máximo de letras
#define MAXLIST 100                                                    // Número máximo de comandos
#define TAM 60


//========================= Mensaje de bienvenida ==========================
void init_shell()
{
    printf("\n\n\n\n======================================================");
    printf("\n\n\n\t****MINIBASH****");
    printf("\n\n\n\n======================================================");

    char* username = getenv("USER");
    printf("\n");
    sleep(1);
}
//========================= Manejo de archivos "<" y " >" ==========================

void redirSal(char cad[TAM]){
    char *cadPtr;
    cadPtr=cad;                                                        //puntero a la cadena
    close(1);                                                          //cerramos la salida estándar
    open(cadPtr,O_CREAT | O_WRONLY,0777);                              //Se asigna la salida al fichero, tambien se le asignan permisos totales

}
void redirEnt(char cad[TAM]){
    char *cadPtr;
    int f;
    cadPtr = cad;                                                      //puntero a la cadena
    f=open(cadPtr,O_RDONLY);                                           // se asigna la salida al fichero
    close(0);                                                          //cerramos la salida estándar
    dup(f);
}
//========================= Lectura de entrada ==========================

int takeInput(char* str)
{
    char* buf;
    buf = readline(">>> ");
    if (strlen(buf) != 0) {
        add_history(buf);
        strcpy(str, buf);
        return 0;
    } else {
        return 1;
    }
}
//========================= Imprime la dirección ==========================

void printDir()
{
    char cwd[1024];
    getcwd(cwd, sizeof(cwd));
    printf("\nDir: %s", cwd);
}
//========================= Ejecuta comandos ==========================


void execArgs(char** parsed)
{

    pid_t pid = fork();
    if (pid == -1) {
        printf("\nFailed forking child..");
        return;
    } else if (pid == 0) {
        if (execvp(parsed[0], parsed) < 0) {
            printf("\nCould not execute command..");
        }
        exit(0);
    } else {                                                           // Esperando al hijo
        wait(NULL);
        return;
```

```
93      }
94  }
95  //========================= Ejecuta comandos en tuberías
        ==========================================================

96
97  void execArgsPiped(char** parsed, char** parsedpipe)
98  {
99      int pipefd[2];
100     pid_t p1, p2;
101     if (pipe(pipefd) < 0) {
102         printf("\nPipe could not be initialized");
103         return;
104     }
105     p1 = fork();
106     if (p1 < 0) {
107         printf("\nCould not fork");
108         return;
109     }
110     if (p1 == 0) {
111                                                                  // Hijo 1
112         close(pipefd[0]);
113         dup2(pipefd[1], STDOUT_FILENO);
114         close(pipefd[1]);
115         if (execvp(parsed[0], parsed) < 0) {
116             printf("\nCould not execute command 1..");
117             exit(0);
118         }
119
120     } else {
121                                                                  //Código del padre
122         p2 = fork();
123         if (p2 < 0) {
124             printf("\nCould not fork");
125             return;
126         }
127                                                                  // Hijo 2
128
129         if (p2 == 0) {
130             close(pipefd[1]);
131             dup2(pipefd[0], STDIN_FILENO);
132             close(pipefd[0]);
133             if (execvp(parsedpipe[0], parsedpipe) < 0) {
134                 printf("\nCould not execute command 2..");
135                 exit(0);
136             }
137         } else {
138                                                                  //Espera a los hijos
139             wait(NULL);
140             wait(NULL);
141         }
142     }
143 }
144 //========================= Manejo de tuberias
        ==========================================================
145 void pipeline(char ***cmd)
146 {
147     int fd[2];
148     pid_t pid;
149     int fdd = 0;
150
151     while (*cmd != NULL) {
152         pipe(fd);
153         if ((pid = fork()) == -1) {
154             perror("fork");
155             exit(1);
156         }
157         else if (pid == 0) {
158             dup2(fdd, 0);
159             if (*(cmd + 1) != NULL) {
160                 dup2(fd[1], 1);
161             }
162             close(fd[0]);
163             execvp((*cmd)[0], *cmd);
164             exit(1);
165         }
166         else {
167             wait(NULL);
168             close(fd[1]);
169             fdd = fd[0];
170             cmd++;
171         }
172     }
173 }
174
175 //========================= Manejo de comandos
        ==========================================================
176
177 int ownCmdHandler(char** parsed)
```

```
{
    int NoOfOwnCmds = 2, i, switchOwnArg = 0;
    char* ListOfOwnCmds[NoOfOwnCmds];
    char* username;
    ListOfOwnCmds[0] = "exit";
    ListOfOwnCmds[1] = "cd";
    for (i = 0; i < NoOfOwnCmds; i++) {
        if (strcmp(parsed[0], ListOfOwnCmds[i]) == 0) {
            switchOwnArg = i + 1;
            break;
        }
    }
    switch (switchOwnArg) {
    case 1:
        exit(0);
    case 2:
        chdir(parsed[1]);
        return 1;

    default:
        break;
    }
    return 0;
}

//========================== Manejo de cadena (tubería) ==========================

int parsePipe(char* str, char** strpiped)
{
    int i = 0;

    while( (strpiped[i] = strsep(&str,"|")) != NULL ){

        i++;
    }

    if (strpiped[1] == NULL)
        return 0;                                   //Regresa cero si no encuntra una tubería
    else {
        return i;
    }
}

 //========================== Manejo de cadena espacios ==========================

void parseSpace(char* str, char** parsed)
{
    int i;
    char *aux;
    for (i = 0; i < MAXLIST; i++) {
        aux = strsep(&str, " ");
        if (aux == NULL){
            parsed[i] = aux;

            break;
        }
        else if (strlen(aux) == 0)
            i--;
        else{
            parsed[i] = aux;
        }
    }
}
//========================== Manejo de cadena espacios tubería ==========================

char ** parseSpacePipes(char* str, char** parsed)
{
    char *aux;
    char **parsed2;
    parsed2 = (char**)malloc(sizeof(char*) * 100);
    for (int i = 0; i < MAXLIST; i++) {
        aux = strsep(&str, " ");
        if (aux == NULL){
            parsed[i] = aux;
            parsed2[i] = aux;
            return parsed2;
        }
        else if (strlen(aux) == 0)
            i--;
        else{
            parsed[i] = aux;
            parsed2[i] = aux;
        }
    }
```

```
}



//========================= Procesar la cadena PRINCIPAL=============================================

int processString(char* str, char** parsed, char** parsedpipe)
{
    char* strpiped[100];
    int piped = 0, i = 0, y, k = 0, ejecutar = 0;
    piped = parsePipe(str, strpiped);
    char entrada[MAXLIST], salida[TAM];

    char ** cmd[(piped+1)];

    if (piped) {

        for(int j = 0; j < piped; j++){
        i = 0;
        while(i <= strlen(strpiped[j])){


            if(strpiped[j][i] =='<'){                                   // si encuentra un simbolo de redireccionamiento de entrada, entrara en el if
                    strpiped[j][i] = ' ';
                    i++;
                    if(strpiped[j][i] !=' '){                           //debe de haber separaciones entre cada letra o simbolo
                        ejecutar=1;
                    }else{
                        i++;                                            //se lee despues del espacio
                            for(y = 0; strpiped[j][i] !='\0' && strpiped[j][i] !=' ' && strpiped[j][i] !='|' && strpiped[j][i] !='>'; y++){
                                entrada[y] = strpiped[j][i];                //vamos formando el argumento
                                strpiped[j][i] = ' ';
                                i++;
                            }
                            entrada[y]='\0';                            // se asigna un terminador de cadena
                            if(strpiped[j][i]!='\0') i++;               //avanzamos a lo que sigue del comando
                            redirEnt(entrada);                          //mandamos el argumento a la funcion para que se pueda procesar el fichero que se abrirá
                        }
            }

            if (strpiped[j][i] == '>') {                                // si encuentra un > corta la cadena que será el fichero que se usará para la salida
                    strpiped[j][i] = ' ';
                    i++;
                    if (strpiped[j][i] != ' '){
                        ejecutar=1;                                     //esto es para confirmar posteriormente que existe un erros de sintaxis
                    }else{
                            i++;                                        // se lee despues del espacio
                                for(y = 0; strpiped[j][i] != '\0';y++){
                                    salida[y] = strpiped[j][i];             // vamos formando el argumento de la funcion redirSal que será el fichero de salida
                                    strpiped[j][i] = ' ';
                                    i++;
                                }
                            salida[y] = '\0';                           // se le asigana el terminador de cadena
                            redirSal(salida);                           //mandamos el argumento creado a la funcion para que sea tomado como el fichero de salida
                    }
            }

            i++;

        }
    }

    if(ejecutar!=0) printf("Error en la sintáxis\n");

        parseSpace(strpiped[0], parsed);


        for(i = 0; i < piped; i++){
            cmd[i] = parseSpacePipes(strpiped[i], parsedpipe);

        }
        cmd[i] = NULL;


```

```
336     } else {
337
338         i = 0;
339         while(i <= strlen(str)){
340             if(str[i] =='<'){                                   // si encuentra un simbolo
    de redireccionamiento de entrada, entrara en el if
341                 str[i] = ' ';
342                 i++;
343                 if(str[i] !=' '){                               //debe de haber
    separaciones entre cada letra o simbolo
344                     ejecutar=1;
345                 }else{
346                     i++;                                        //se lee despues del
    espacio
347                         for(y = 0; str[i] !='\0' && str[i] !=' ' && str[i] !='|' && str[i]
    !='>'; y++){
348                             entrada[y] = str[i];                //vamos formando el
    argumento
349                             str[i] = ' ';
350                             i++;
351                         }
352                         entrada[y]='\0';                        // se asigna un terminador
    de cadena
353                         if(str[i]!='\0') i++;                   //avanzamos a lo que sigue
    del comando
354                         redirEnt(entrada);                      //mandamos el argumento a
    la funcion para que se pueda procesar el fichero que se abrirá
355                 }
356             }
357
358             if (str[i] == '>') {                                // si encuentra un > corta
    la cadena que será el fichero que se usará para la salida
359                 str[i] = ' ';
360                 i++;
361                 if (str[i] != ' '){
362                     ejecutar=1;                                //esto es para confirmar
    posteriormente que existe un erros de sintaxis
363                 }else{
364                     i++;                                       // se lee despues del espacio
365                         for(y = 0; str[i] != '\0';y++){
366                             salida[y] = str[i];                 // vamos formando el
    argumento de la funcion redirSal que será el fichero de salida
367                             str[i] = ' ';
368                             i++;
369                         }
370                     salida[y] = '\0';                          // se le asigana el
    terminador de cadena
371                     redirSal(salida);                          //mandamos el argumento
    creado a la funcion para que sea tomado como el fichero de salida
372                 }
373             }
374
375             i++;
376
377         }
378
379          if(ejecutar!=0) printf("Error en la sintáxis\n");
380
381
382         parseSpace(str, parsed);
383
384     }
385     if (ownCmdHandler(parsed))
386         return 0;
387     else{
388         pipeline(cmd);
389         return 1 + piped;
390     }
391 }
392
393 int main()
394 {
395
396     char inputString[MAXCOM], *parsedArgs[MAXLIST];
397     char* parsedArgsPiped[MAXLIST];
398     int execFlag = 0;
399     int stdout = dup(1), stdin = dup(0);
400
401     init_shell();
402
403     while (1) {
404         close(1);                                              // Se cierra la salida que
    tenga
405         dup(stdout);
406             close(0);                                          //Se cierra la salida,
    tambien cierra el fichero cuando se ha guardado en el
407         dup(stdin);
408
```

```
409         printDir();                                                    //imprime
410
411         if (takeInput(inputString))
412             continue;
413
414         execFlag = processString(inputString, parsedArgs, parsedArgsPiped);
415
416         if (execFlag == 1)
417             execArgs(parsedArgs);                                      //Ejecuta si hay comandos
418
419     }
420     return 0;
421 }
```