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.

3. Desarrollo

La manera en que procesaremos los comandos ingresados en nuestro minibash

- 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

Anexos

```
1 /*========================================================================*/
2 /*=========================== INSTITUTO POLITÉCNICO NACIONAL =============*/
3 /*============================ ESCUELA SUPERIOR DE CÓMPUTO ===============*/
4 /*========================================================================*/
5 /*============================== Ivan Aldavera Gallaga ===================*/
6 /*============================== Erick Francisco Vázquez Nuñez ===========*/
```

```c
/*============== Laura Andrea Morales López ==============*/
/*=========================================================*/

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

```
84          return;
85      } else if (pid == 0) {
86          if (execvp(parsed[0], parsed) < 0) {
87              printf("\nCould not execute command..");
88          }
89          exit(0);
90      } else {
91                                                          // Esperando al hijo
92          wait(NULL);
93          return;
94      }
95  }
96  //================================ Ejecuta comandos en tuberías ================================
97
98  void execArgsPiped(char** parsed, char** parsedpipe)
99  {
100     int pipefd[2];
101     pid_t p1, p2;
102     if (pipe(pipefd) < 0) {
103         printf("\nPipe could not be initialized");
104         return;
105     }
106     p1 = fork();
107     if (p1 < 0) {
108         printf("\nCould not fork");
109         return;
110     }
111     if (p1 == 0) {
112                                                          // Hijo 1
113         close(pipefd[0]);
114         dup2(pipefd[1], STDOUT_FILENO);
115         close(pipefd[1]);
116         if (execvp(parsed[0], parsed) < 0) {
117             printf("\nCould not execute command 1..");
118             exit(0);
119         }
120
121     } else {
122                                                          //Código del padre
123         p2 = fork();
124         if (p2 < 0) {
125             printf("\nCould not fork");
126             return;
127         }
128                                                          // Hijo 2
129
130         if (p2 == 0) {
131             close(pipefd[1]);
132             dup2(pipefd[0], STDIN_FILENO);
133             close(pipefd[0]);
134             if (execvp(parsedpipe[0], parsedpipe) < 0) {
135                 printf("\nCould not execute command 2..");
136                 exit(0);
137             }
138         } else {
139                                                          //Espera a los hijos
140             wait(NULL);
141             wait(NULL);
142         }
143     }
144 }
145 //================================ Manejo de tuberias ================================
146 void pipeline(char ***cmd)
147 {
148     int fd[2];
149     pid_t pid;
150     int fdd = 0;
151
152     while (*cmd != NULL) {
153         pipe(fd);
154         if ((pid = fork()) == -1) {
155             perror("fork");
156             exit(1);
157         }
158         else if (pid == 0) {
159             dup2(fdd, 0);
160             if (*(cmd + 1) != NULL) {
161                 dup2(fd[1], 1);
162             }
163             close(fd[0]);
164             execvp((*cmd)[0], *cmd);
165             exit(1);
166         }
167         else {
168             wait(NULL);
169             close(fd[1]);
```

```
170             fdd = fd[0];
171             cmd++;
172         }
173     }
174 }
175
176 //========================== Manejo de comandos ==========================
177
178 int ownCmdHandler(char** parsed)
179 {
180     int NoOfOwnCmds = 2, i, switchOwnArg = 0;
181     char* ListOfOwnCmds[NoOfOwnCmds];
182     char* username;
183     ListOfOwnCmds[0] = "exit";
184     ListOfOwnCmds[1] = "cd";
185     for (i = 0; i < NoOfOwnCmds; i++) {
186         if (strcmp(parsed[0], ListOfOwnCmds[i]) == 0) {
187             switchOwnArg = i + 1;
188             break;
189         }
190     }
191     switch (switchOwnArg) {
192     case 1:
193         exit(0);
194     case 2:
195         chdir(parsed[1]);
196         return 1;
197
198     default:
199         break;
200     }
201     return 0;
202 }
203
204 //========================== Manejo de tubería ==========================
205
206 int parsePipe(char* str, char** strpiped)
207 {
208     int i = 0;
209
210     while( (strpiped[i] = strsep(&str,"|")) != NULL ){
211
212         i++;
213     }
214
215     if (strpiped[1] == NULL)
216         return 0;                                   //Regresa cero si
        no encuntra una tubería
217     else {
218         return i;
219     }
220 }
221
222  //========================== Manejo de cadena ==========================
223
224 void parseSpace(char* str, char** parsed)
225 {
226     int i;
227     char *aux;
228     for (i = 0; i < MAXLIST; i++) {
229         aux = strsep(&str, " ");
230         if (aux == NULL){
231             parsed[i] = aux;
232
233             break;
234         }
235         else if (strlen(aux) == 0)
236             i--;
237         else{
238             parsed[i] = aux;
239         }
240     }
241 }
242 //========================== Manejo de cadena
        tubería==========================
243
244 char ** parseSpacePipes(char* str, char** parsed)
245 {
246     char *aux;
247     char **parsed2;
248     parsed2 = (char**)malloc(sizeof(char*) * 100);
249     for (int i = 0; i < MAXLIST; i++) {
250         aux = strsep(&str, " ");
251         if (aux == NULL){
252             parsed[i] = aux;
```

```
253             parsed2[i] = aux;
254             return parsed2;
255         }
256         else if (strlen(aux) == 0)
257             i--;
258         else{
259             parsed[i] = aux;
260             parsed2[i] = aux;
261         }
262     }
263 }
264 //========================= Procesar la cadena ==========================
265
266
267
268 int processString(char* str, char** parsed, char** parsedpipe)
269 {
270     char* strpiped[100];
271     int piped = 0, i = 0, y, k = 0, ejecutar = 0;
272     piped = parsePipe(str, strpiped);
273     char entrada[MAXLIST], salida[TAM];
274
275     char ** cmd[(piped+1)];
276
277     if (piped) {
278
279         for(int j = 0; j < piped; j++){
280         i = 0;
281         while(i <= strlen(strpiped[j])){
282
283
284             if(strpiped[j][i] =='<'){                                   // si encuentra un simbolo de redireccionamiento de entrada, entrara en el if
285                     strpiped[j][i] = ' ';
286                     i++;
287                     if(strpiped[j][i] !=' '){                          //debe de haber separaciones entre cada letra o simbolo
288                         ejecutar=1;
289                     }else{
290                         i++;                                           //se lee despues del espacio
291                             for(y = 0; strpiped[j][i] !='\0' && strpiped[j][i] !=' ' && strpiped[j][i] !='|' && strpiped[j][i] !='>'; y++){
292                                 entrada[y] = strpiped[j][i];           //vamos formando el argumento
293                                 strpiped[j][i] = ' ';
294                                 i++;
295                             }
296                             entrada[y]='\0';                           // se asigna un terminador de cadena
297                             if(strpiped[j][i]!='\0') i++;              //avanzamos a lo que sigue del comando
298                             redirEnt(entrada);                         //mandamos el argumento a la funcion para que se pueda procesar el fichero que se abrirá
299                         }
300             }
301
302             if (strpiped[j][i] == '>') {                               // si encuentra un > corta la cadena que será el fichero que se usará para la salida
303                     strpiped[j][i] = ' ';
304                     i++;
305                     if (strpiped[j][i] != ' '){
306                         ejecutar=1;                                    //esto es para confirmar posteriormente que existe un erros de sintaxis
307                     }else{
308                             i++;                                       // se lee despues del espacio
309                                 for(y = 0; strpiped[j][i] != '\0';y++){
310                                     salida[y] = strpiped[j][i];        // vamos formando el argumento de la funcion redirSal que será el fichero de salida
311                                     strpiped[j][i] = ' ';
312                                     i++;
313                                 }
314                             salida[y] = '\0';                          // se le asigana el terminador de cadena
315                             redirSal(salida);                          //mandamos el argumento creado a la funcion para que sea tomado como el fichero de salida
316                     }
317             }
318
319             i++;
320
321         }
322     }
323
324     if(ejecutar!=0) printf("Error en la sintáxis\n");
325
326         parseSpace(strpiped[0], parsed);
```

```
327
328
329         for(i = 0; i < piped; i++){
330             cmd[i] = parseSpacePipes(strpiped[i], parsedpipe);
331
332         }
333         cmd[i] = NULL;
334
335
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 }
```