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

3. Desarrollo

4. Resultados

5. Conclusiones

Anexos

```c
// C Program to design a shell in Linux
#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 // max number of letters to be supported
#define MAXLIST 100 // max number of commands to be supported
#define TAM 60
// Clearing the shell using escape sequences
#define clear() printf("\033[H\033[J")
// Greeting shell during startup
char ** cmdGlobal;


void init_shell()
{
    clear();
    printf("\n\n\n\n******************"
        "************************");
    printf("\n\n\n\t****MY SHELL****");
    printf("\n\n\t-USE AT YOUR OWN RISK-");
    printf("\n\n\n\n*******************"
        "************************");
    char* username = getenv("USER");
    printf("\n\n\nUSER is: @%s", username);
    printf("\n");
    sleep(1);
    clear();
}
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
```

```
41
42  }
43  void redirEnt(char cad[TAM]){
44      char *cadPtr;
45      int f;
46      cadPtr = cad; //puntero a la cadena
47      f=open(cadPtr,O_RDONLY); // se asigna la salida al fichero
48      close(0); //cerramos la salida estándar
49      dup(f);
50  }
51
52  // Function to take input
53  int takeInput(char* str)
54  {
55      char* buf;
56      buf = readline(">>> ");
57      if (strlen(buf) != 0) {
58          add_history(buf);
59          strcpy(str, buf);
60          return 0;
61      } else {
62          return 1;
63      }
64  }
65
66  // Function to print Current Directory.
67  void printDir()
68  {
69      char cwd[1024];
70      getcwd(cwd, sizeof(cwd));
71      printf("\nDir: %s", cwd);
72  }
73
74  // Function where the system command is executed
75  void execArgs(char** parsed)
76  {
77      // Forking a child
78      pid_t pid = fork();
79      if (pid == -1) {
80          printf("\nFailed forking child..");
81          return;
82      } else if (pid == 0) {
83          if (execvp(parsed[0], parsed) < 0) {
84              printf("\nCould not execute command..");
85          }
86          exit(0);
87      } else {
88          // waiting for child to terminate
89          wait(NULL);
90          return;
91      }
92  }
93
94  // Function where the piped system commands is executed
95  void execArgsPiped(char** parsed, char** parsedpipe)
96  {
97      // 0 is read end, 1 is write end
98      int pipefd[2];
99      pid_t p1, p2;
100     if (pipe(pipefd) < 0) {
101         printf("\nPipe could not be initialized");
102         return;
103     }
104     p1 = fork();
105     if (p1 < 0) {
106         printf("\nCould not fork");
107         return;
108     }
109     if (p1 == 0) {
110         // Child 1 executing..
111         // It only needs to write at the write end
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         // Parent executing
122         p2 = fork();
123         if (p2 < 0) {
124             printf("\nCould not fork");
125             return;
126         }
127         // Child 2 executing..
128         // It only needs to read at the read end
```

```
        if (p2 == 0) {
            close(pipefd[1]);
            dup2(pipefd[0], STDIN_FILENO);
            close(pipefd[0]);
            if (execvp(parsedpipe[0], parsedpipe) < 0) {
                printf("\nCould not execute command 2..");
                exit(0);
            }
        } else {
            // parent executing, waiting for two children
            wait(NULL);
            wait(NULL);
        }
    }
}

void pipeline(char ***cmd)
{
    int fd[2];
    pid_t pid;
    int fdd = 0;                /* Backup */

    while (*cmd != NULL) {
        pipe(fd);               /* Sharing bidiflow */
        if ((pid = fork()) == -1) {
            perror("fork");
            exit(1);
        }
        else if (pid == 0) {
            dup2(fdd, 0);
            if (*(cmd + 1) != NULL) {
                dup2(fd[1], 1);
            }
            close(fd[0]);
            execvp((*cmd)[0], *cmd);
            exit(1);
        }
        else {
            wait(NULL);         /* Collect childs */
            close(fd[1]);
            fdd = fd[0];
            cmd++;
        }
    }
}

// Help command builtin
void openHelp()
{
    puts("\n***WELCOME TO MY SHELL HELP***"
        "\nCopyright @ Suprotik Dey"
        "\n-Use the shell at your own risk..."
        "\nList of Commands supported:"
        "\n>cd"
        "\n>ls"
        "\n>exit"
        "\n>all other general commands available in UNIX shell"
        "\n>pipe handling"
        "\n>improper space handling");
    return;
}

// Function to execute builtin commands
int ownCmdHandler(char** parsed)
{
    int NoOfOwnCmds = 4, i, switchOwnArg = 0;
    char* ListOfOwnCmds[NoOfOwnCmds];
    char* username;
    ListOfOwnCmds[0] = "exit";
    ListOfOwnCmds[1] = "cd";
    ListOfOwnCmds[2] = "help";
    ListOfOwnCmds[3] = "hello";
    for (i = 0; i < NoOfOwnCmds; i++) {
        if (strcmp(parsed[0], ListOfOwnCmds[i]) == 0) {
            switchOwnArg = i + 1;
            break;
        }
    }
    switch (switchOwnArg) {
    case 1:
        printf("\nGoodbye\n");
        exit(0);
    case 2:
        chdir(parsed[1]);
        return 1;
    case 3:
        openHelp();
        return 1;
```

```
217     case 4:
218         username = getenv("USER");
219         printf("\nHello %s.\nMind that this is "
220             "not a place to play around."
221             "\nUse help to know more..\n",
222             username);
223         return 1;
224     default:
225         break;
226     }
227     return 0;
228 }
229
230 // function for finding pipe
231 int parsePipe(char* str, char** strpiped)
232 {
233     int i = 0;
234    /* for (i = 0; i < 2; i++) {
235         strpiped[i] = strsep(&str, "|");
236         if (strpiped[i] == NULL)
237             break;
238     } */
239
240     while( (strpiped[i] = strsep(&str,"|")) != NULL ){
241
242        // printf("%s\n",strpiped[i]);
243         i++;
244     }
245
246     if (strpiped[1] == NULL)
247         return 0; // returns zero if no pipe is found.
248     else {
249         return i;
250     }
251 }
252
253 // function for parsing command words
254 void parseSpace(char* str, char** parsed)
255 {
256     int i;
257     char *aux;
258     for (i = 0; i < MAXLIST; i++) {
259         aux = strsep(&str, " ");
260         if (aux == NULL){
261             parsed[i] = aux;
262          //    printf("\n---- %s\n", parsed[i]);
263             //reurn parsed;
264             break;
265         }
266         else if (strlen(aux) == 0)
267             i--;
268         else{
269             parsed[i] = aux;
270          //    printf("\n---- %s\n", parsed[i]);
271         }
272     }
273 }
274
275 char ** parseSpacePipes(char* str, char** parsed)
276 {
277     char *aux;
278     char **parsed2;
279     parsed2 = (char**)malloc(sizeof(char*) * 100);
280     for (int i = 0; i < MAXLIST; i++) {
281         aux = strsep(&str, " ");
282         if (aux == NULL){
283             parsed[i] = aux;
284             parsed2[i] = aux;
285            // printf("\n---- %s\n", parsed2[i]);
286             return parsed2;
287         }
288         else if (strlen(aux) == 0)
289             i--;
290         else{
291             parsed[i] = aux;
292             parsed2[i] = aux;
293            // printf("\n---- %s\n", parsed2[i]);
294         }
295     }
296 }
297
298
299
300 int processString(char* str, char** parsed, char** parsedpipe)
301 {
302     char* strpiped[100];
303     int piped = 0, i = 0, y, k = 0, ejecutar = 0;
304     piped = parsePipe(str, strpiped);
```

```
305     char entrada[MAXLIST], salida[TAM];
306
307
308    // printf("\nse detectaron %d tuberias\n",piped);
309
310
311
312     char ** cmd[(piped+1)];
313
314     //char ** cmdAux[piped+1];
315
316 /*
317     char *ls[] = {"ls", NULL};
318     char *sort[] = {"sort", "-n", NULL};
319     char *wc[] = {"wc", NULL};
320
321
322     char ** comandos[] = {ls, sort, wc, NULL};
323
324     for(i = 0; i < 4; i++){
325         cmdAux[i] = comandos[i];
326     }
327
328     for(i = 0; i < 3; i++){
329             printf("\npalabra %s en la posicion %d de cmdAux\n", cmdAux[i][0], i);
330         }
331 */
332     /*for(i = 0; i < piped; i++){
333             printf("\npalabra%sen la posicion %d\n", strpiped[i], i);
334         }*/
335
336     if (piped) {
337
338         for(int j = 0; j < piped; j++){
339         i = 0;
340         while(i <= strlen(strpiped[j])){
341
342                 //  printf("%c\n", strpiped[j][i]);
343             if(strpiped[j][i] =='<'){ // si encuentra un simbolo de redireccionamiento de entrada,
    entrara en el if
344                     strpiped[j][i] = ' ';
345                     i++;
346                     if(strpiped[j][i] !=' '){ //debe de haber separaciones entre cada letra o
    simbolo
347                         ejecutar=1;
348                     }else{
349                         i++; //se lee despues del espacio
350                             for(y = 0; strpiped[j][i] !='\0' && strpiped[j][i] !=' ' &&
    strpiped[j][i] !='|' && strpiped[j][i] !='>'; y++){
351                                 entrada[y] = strpiped[j][i]; //vamos formando el argumento
352                                 strpiped[j][i] = ' ';
353                                 i++;
354                             }
355                             entrada[y]='\0'; // se asigna un terminador de cadena
356                             if(strpiped[j][i]!='\0') i++; //avanzamos a lo que sigue del comando
357                             redirEnt(entrada); //mandamos el argumento a la funcion para que se
    pueda procesar el fichero que se abrirá
358                     }
359             }
360
361             if (strpiped[j][i] == '>') { // si encuentra un > corta la cadena que será el fichero
    que se usará para la salida
362                     strpiped[j][i] = ' ';
363                     i++;
364                     if (strpiped[j][i] != ' '){
365                         ejecutar=1; //esto es para confirmar posteriormente que existe un erros de
    sintaxis
366                     }else{
367                             i++; // se lee despues del espacio
368                                 for(y = 0; strpiped[j][i] != '\0';y++){
369                                     salida[y] = strpiped[j][i]; // vamos formando el argumento de
    la funcion redirSal que será el fichero de salida
370                                     strpiped[j][i] = ' ';
371                                     i++;
372                                 }
373                         salida[y] = '\0'; // se le asigana el terminador de cadena
374                         redirSal(salida); //mandamos el argumento creado a la funcion para que sea
    tomado como el fichero de salida
375                     }
376             }
377
378             i++;
379
380         }
381     }
382
383     if(ejecutar!=0) printf("Error en la sintáxis\n");
384
```

```
385         parseSpace(strpiped[0], parsed);
386 
387 
388         for(i = 0; i < piped; i++){
389             // printf("\npalabras que se van a guardar: %s en la posicion %d\n",
    *parseSpacePipes(strpiped[i], parsedpipe), i);
390 
391             cmd[i] = parseSpacePipes(strpiped[i], parsedpipe);
392            // printf("\npalabra %s en la posicion %d que es la nueva posicion\n", cmd[j][0], j);
393             //if(j > 0)
394               //   printf("\npalabra %s en la posicion %d que es la anterior a la actual\n",
    cmd[j-1][0], j-1);
395         }
396        // printf("\n\nnum : %d\n\n", i);
397         cmd[i] = NULL;
398 
399       //  printf("\npalabra %s \n", cmd[0][0]);
400       //  printf("\npalabra %s \n", cmd[1][1]);
401       //  printf("\npalabra %s \n", cmd[2][0]);
402       //  printf("\npalabra %s \n", cmd[3]);
403 
404        /*for(i = 0; i < piped+1; i++){
405             printf("\npalabra %s en la posicion %d\n", cmd[i][0], i);
406         }*/
407        // printf("\n\nsi pasamos de qui como no\n\n");
408 
409         //parseSpace(strpiped[1], parsedpipe);
410     } else {
411 
412         i = 0;
413         while(i <= strlen(str)){
414               //  printf("%c\n", strpiped[j][i]);
415             if(str[i] =='<'){ // si encuentra un simbolo de redireccionamiento de entrada, entrara
    en el if
416                     str[i] = ' ';
417                     i++;
418                     if(str[i] !=' '){ //debe de haber separaciones entre cada letra o simbolo
419                         ejecutar=1;
420                     }else{
421                         i++; //se lee despues del espacio
422                             for(y = 0; str[i] !='\0' && str[i] !=' ' && str[i] !='|' && str[i]
    !='>'; y++){
423                                 entrada[y] = str[i]; //vamos formando el argumento
424                                 str[i] = ' ';
425                                 i++;
426                             }
427                             entrada[y]='\0'; // se asigna un terminador de cadena
428                             if(str[i]!='\0') i++; //avanzamos a lo que sigue del comando
429                             redirEnt(entrada); //mandamos el argumento a la funcion para que se
    pueda procesar el fichero que se abrirá
430                     }
431             }
432 
433             if (str[i] == '>') { // si encuentra un > corta la cadena que será el fichero que se
    usará para la salida
434                     str[i] = ' ';
435                     i++;
436                     if (str[i] != ' '){
437                         ejecutar=1; //esto es para confirmar posteriormente que existe un erros de
    sintaxis
438                     }else{
439                             i++; // se lee despues del espacio
440                                 for(y = 0; str[i] != '\0';y++){
441                                     salida[y] = str[i]; // vamos formando el argumento de la
    funcion redirSal que será el fichero de salida
442                                     str[i] = ' ';
443                                     i++;
444                                 }
445                         salida[y] = '\0'; // se le asigana el terminador de cadena
446                         redirSal(salida); //mandamos el argumento creado a la funcion para que sea
    tomado como el fichero de salida
447                     }
448             }
449 
450             i++;
451 
452         }
453 
454          if(ejecutar!=0) printf("Error en la sintáxis\n");
455 
456 
457         parseSpace(str, parsed);
458 
459     }
460     if (ownCmdHandler(parsed))
461         return 0;
462     else{
463         pipeline(cmd);
```

```
464         return 1 + piped;
465     }
466 }
467
468 int main()
469 {
470
471     char inputString[MAXCOM], *parsedArgs[MAXLIST];
472     char* parsedArgsPiped[MAXLIST];
473     int execFlag = 0;
474     int stdout = dup(1), stdin = dup(0);
475
476     init_shell();
477
478     while (1) {
479         close(1); // Se cierra la salida que tenga
480         dup(stdout);
481             close(0); //Se cierra la salida, tambien cierra el fichero cuando se ha guardado en el
482         dup(stdin);
483         // print shell line
484         printDir();
485         // take input
486         if (takeInput(inputString))
487             continue;
488         // process
489         execFlag = processString(inputString, parsedArgs, parsedArgsPiped);
490         // execflag returns zero if there is no command
491         // or it is a builtin command,
492         // 1 if it is a simple command
493         // 2 if it is including a pipe.
494        // printf("\n result: %d\n", execFlag);
495         // execute
496         if (execFlag == 1)
497             execArgs(parsedArgs);
498         //if (execFlag > 1)
499           //  execArgsPiped(parsedArgs, parsedArgsPiped);
500     }
501     return 0;
502 }
```