Alumno:

Benitez Miranda Samuel Eduardo

Unidad Académica:

Sistemas Operativos (2CM9)

Profesora:

Juárez Méndez Ana Belem

Proyecto:

MiniShell

Fecha de entrega:

18 de mayo de 2020

Objetivo

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

Introducción

¿Qué es UNIX?

UNIX es un sistema operativo, es decir una colección de programas que ejecutan otros programas en una computadora. Este sistema operativo, se caracteriza por ser portable y multitarea.

Este sistema provee una serie de herramientas, cada una realiza una función limitada y bien definida, utiliza un sistema de archivos unificado como medio de comunicación y un lenguaje de comandos llamado Shell.

Hoy en día, los sistemas operativos UNIX son ampliamente utilizados en multitud de dispositivos que abarcan desde los supercomputadores más capaces hasta los teléfonos móviles más populares, pasando por los ordenadores que utilizamos diariamente en nuestros escritorios.

¿Qué es Shell?

Una Shell de Unix o también shell, es un intérprete de comandos, el cual consiste en la interfaz de usuario tradicional de los sistemas operativos basados en Unix y similares, como GNU/Linux.

Mediante las instrucciones que aporta el intérprete, el usuario puede comunicarse con el núcleo y por extensión, ejecutar dichas órdenes, así como herramientas que le permiten controlar el funcionamiento de la computadora.

Los comandos que aportan los intérpretes, pueden usarse a modo de guion si se escriben en ficheros ejecutables denominados shell-scripts, de este modo, cuando el usuario necesita hacer uso de varios comandos o combinados de comandos con herramientas, escribe en un fichero de texto, marcado como ejecutable, las operaciones que posteriormente, línea por línea, el intérprete traducirá al núcleo para que las realice.

Debe trabajar con el agrupamiento de nombres de archivo, tuberías, here documents, redirecciones, sustitución de comandos, variables y estructuras de control para pruebas de condición e iteración.

Existen varios tipos de shell, el más común es Bash encontrada en sistemas GNU/Linux.

Desarrollo

El proyecto consta de un intérprete de comandos, el cual tiene la capacidad de ejecutar dichos comandos o programas con o sin parámetros, pipes y redireccionamientos de entrada y salida (|, <, >, >>). Es decir, debe poder ejecutar comandos de la forma:

- *Sin argumentos:* ls, which, pwd, ps, wc, pstree…
- *Con argumentos:* cal -m 2, ls -l -a, cp archivo1 archivo2…
- *Con pipes:* ls | wc | wc, ps -a | sort…
- *Con redireccionamientos:* ls > archivo, wc < archivo, rev < archivo1 > archivo2…
- *Mezclas de los anteriores:* ls | sort -n | wc > archivo

Solución y Código

```
typedef struct {
     char * nombreComando;
     char ** argv;
} comando;

typedef struct {
     int nComandos;
     comando * comandos;
     char * entrada;
     int concatenar;
     char * salida;
} linea;
```

Primeramente, se crearon dos estructuras que nos permitirán almacenar la información necesaria de la línea de comandos, una de tipo *comando*, en la cual se almacenarán el nombre de un comando en particular, así como un arreglo de argumentos.

Por otro lado, otra estructura de tipo *línea*, la cual almacenará toda la información de la entrada ingresada por el usuario en la línea de comandos:

- *Número de comandos:* cantidad de comandos identificados en la entrada
- *Arreglo de comandos:* todos los comandos con sus respectivos nombres y argumentos
- *Redirección de entrada:* si existe un operador '<' se almacenará el siguiente argumento como un archivo de entrada
- *Bandera para concatenar en el archivo:* permitirá identificar el modo de apertura del archivo de salida.
- *Redirección de salida:* si existe un operador '>' o '>>' se almacenará el siguiente argumento como archivo de salida.

Funciones

```
void ejecutar(char* tokens[]);
void parse(char *linea, char **tokens);
void imprimeTokens(char* tokens[]);
int numeroComandos(char* tokens[]);
bool esComando(char* token);
void Salida(char* tokens[], linea* instruccion);
void Entrada(char* tokens[], linea* instruccion);
```

Cada una de las siguientes funciones ayudarán a procesar la línea ingresada por el usuario, se describirá su uso individual y como se comunican entre sí para llevar a cabo la ejecución de los comandos identificados durante el procesamiento.

void parse (char *línea, char **tokens);

```
void parse(char *linea, char **tokens){
     while (*linea != '\0') {
          while (*linea == ' ' || *linea == '\t' || *linea == '\n')
               *linea++ = '\0';
          *tokens++ = linea;
          while (*linea != '\0' && *linea != ' ' && *linea != '\t' && *linea != '\n')
               linea++;
     }
     *tokens = '\0';  //NULL
}
```

Será la primera función en ejecutarse, puesto que recibirá toda la línea ingresada por el usuario (char *linea) y almacenará cada una de las palabras como tokens separados por espacios o tabulaciones y guardará todos ellos en un arreglo de cadenas (char **tokens).

bool esComando (char* token);

```
bool esComando(char* token){
     for (int i = 0; i < comandosUNIX; i++){
          if(strcmp(token, comandos[i]) == 0){
               return TRUE;
          }
     }
     return FALSE;
}
```

Esta función verificará la existencia de un comando ingresado que se encuentre en el arreglo de tokens. Regresa TRUE (1) si es un comando y FALSE (0) si no lo es.

int numeroComandos (char* tokens [ ]);

```
int numeroComandos(char* tokens[]){
     int numComandos = 0;
     for (int i = 0; i < 63; i++){
          if(tokens[i] == NULL){
               N_TOKENS = i;
               return numComandos;
          }else{
               if(esComando(tokens[i])){
                    numComandos++;
               }
          }
     }
}
```

Esta función determinará el número de comandos identificados en el arreglo que se obtiene mediante *parse( )*, con la ayuda de *esComando( )* comparando cada token con un arreglo estático con los nombres de los comandos más comunes en Unix, que son los siguientes:

```
char* comandos[] = {"exit","clear","cd","ls","wc","cal","cat","cp","date","find","grep","mkdir","mv","ps","pstree",
                    "rev","pwd","sort","uname","wc","which","who","man","df","w","rm","head","echo","whoami"};
int comandosUNIX = sizeof(comandos) / sizeof(comandos[0]);
```

El valor de retorno se almacenará en el atributo *nComandos* de la estructura *línea*.

void Entrada (char* tokens [ ], línea *instruccion);

```
void Entrada(char* tokens[],linea* instruccion){
     for (int i = 0; i < 63; i++){
          if(tokens[i] == NULL){
               instruccion->entrada = NULL;
               return;
          }else{
               if(strcmp(tokens[i], "<") == 0){
                    instruccion->entrada = tokens[i+1];
                    return;
               }
          }
     }
}
```

Esta función recibirá tanto el arreglo de tokens como el apuntador a la estructura de tipo *línea* que se estará construyendo, en el caso en que uno de los tokens sea el operador '<', se guardará el siguiente token del arreglo en el atributo *entrada* de la estructura referenciada por el apuntador.

void Salida (char* tokens [ ], línea *instruccion);

```
void Salida(char* tokens[],linea* instruccion){
     for (int i = 0; i < 63; i++){
          if(tokens[i] == NULL){
               instruccion->concatenar = 0;
               instruccion->salida = NULL;
               return;
          }else{
               if(strcmp(tokens[i], ">") == 0){
                    instruccion->concatenar = 0;
                    instruccion->salida = tokens[i+1];
                    return;
               }else if(strcmp(tokens[i], ">>") == 0){
                    instruccion->concatenar = 1;
                    instruccion->salida = tokens[i+1];
                    return;
               }
          }
     }
}
```

De forma análoga a la función anterior, esta identificará si un token es el operador '>' o '>>', de ser así, se guardará el siguiente token en el atributo *salida* de la estructura referenciada por el apuntador. Sin embargo, si el operador fuese '>>', la bandera de *concatenación* tendría un valor de 1 y servirá para determinar el modo de apertura del archivo en cuestión cuando se ejecuten los comandos.

void imprimeTokens (char* tokens [ ]);

```
void imprimeTokens(char* tokens[]){
     for (int i = 0; i < 63; i++){
          if(tokens[i] != NULL){
               printf("TOKEN [%i] = [%s]\n",i,tokens[i]);
          }else{
               return;
          }
     }
}
```

Función auxiliar que permite monitorear la forma en que se guardaron los tokens según la línea ingresada por el usuario.

void ejecutar (char tokens [ ]);*

Esta función recibirá como único parámetro el arreglo de tokens que haya sido creado por *parse( )* y hará uso de todas las funciones antes mencionadas para ejecutar los comandos.

```
void ejecutar(char* tokens[]){
     int i,j=0,k,pid,*pidHijos,status, fichero,**pipes; //Arreglo de hijos, matriz de pipes
     char ** argvAux1 = (char**)malloc(sizeof(char**)), ** argvAux2 = (char**)malloc(sizeof(char**)), ** argvAux3 = (char**)malloc(sizeof(char**)),
          ** argvAux4 = (char**)malloc(sizeof(char**)), ** argvAux5 = (char**)malloc(sizeof(char**));

     linea* instruccion = (linea*)malloc(sizeof(linea));
     instruccion->nComandos = numeroComandos(tokens);
     comando arrayComandos[5];

     arrayComandos[0].argv = argvAux1;
     arrayComandos[1].argv = argvAux2;
     arrayComandos[2].argv = argvAux3;
     arrayComandos[3].argv = argvAux4;
     arrayComandos[4].argv = argvAux5;

     instruccion->comandos = arrayComandos;

     for(i=0; i< instruccion->nComandos; i++){
          instruccion->comandos[i].argv = arrayComandos[i].argv;
     }

     Entrada(tokens,instruccion);
     Salida(tokens,instruccion);

     printf("\n# Comandos: [%d]\n",instruccion->nComandos);
     printf("\n# N_TOKENS: [%d]\n",N_TOKENS);

     for(i=0;i < instruccion->nComandos; i++){
          if(j < N_TOKENS){
               if(esComando(tokens[j])){
                    instruccion->comandos[i].nombreComando = tokens[j];
                    printf("Comando: [%i]  Nombre: [%s]\n",i,instruccion->comandos[i].nombreComando);
                    k=0;
                    while(strcmp(tokens[j],"|") != 0 && strcmp(tokens[j],">") != 0 && strcmp(tokens[j],">>") != 0 &&
                         strcmp(tokens[j],"<") != 0 && strcmp(tokens[j],"<<") != 0){

                         instruccion->comandos[i].argv[k] = tokens[j];
                         printf("Comando [%i] Argumento [%i]: [%s]\n",i,k,instruccion->comandos[i].argv[k]);
                         //printf("I[%i] J[%i] K[%i]\n",i,j,k);
                         k++; j++;

                         if(j >= N_TOKENS){
                              break;
                         }
                    }
                    if(strcmp(instruccion->comandos[0].nombreComando,"which")==0 || strcmp(instruccion->comandos[0].nombreComando,"man")==0){
                         instruccion->nComandos = 1;
                    }
                    instruccion->comandos[i].argv[k] = NULL;
               }else{
                    i--; j++;
               }
          }
      }

      printf("\n\n");
      i=0;

      if (strcmp(instruccion->comandos[0].argv[0], "exit") == 0){
                exit(0);
           }else if(strcmp(instruccion->comandos[0].argv[0], "cd") == 0){
                int error = chdir(instruccion->comandos[0].argv[1]); // Nos lleva al destino solicitado por el primer argumento

                if (error == -1){ // Si no existe ese destino
                     printf("ERROR: El directorio no existe: %s \n", instruccion->comandos[0].argv[1]);
                }

           /*--------------------------------1 COMANDO--------------------------------*/

           }else if(instruccion->nComandos == 1){ //Si sólo hay un comando

                pid = fork();

                if (pid < 0){     // Crea un proceso hijo
                     fprintf(stderr, "ERROR: fork() fallo: [%s]\n",strerror(errno));
                     exit(-1);
                }else if (pid == 0){
                     if(instruccion->entrada != NULL){ //si hay redirección de entrada
                               int fichero;
                               printf("redirección de entrada: %s\n", instruccion->entrada);
                               fichero = open(instruccion->entrada, O_RDONLY);
                               if (fichero == -1) {
                                    fprintf(stderr,"%i: Error. Fallo al abrir el fichero de redirección de entrada\n", fichero);
                                    exit(1);
                               } else {
                                    dup2(fichero,0); //redirijimos la entrada
                               }
                          if (instruccion->salida != NULL) { //si hay redirección de salida
                               int fichero2;

                               if(instruccion->concatenar == 1){ //si hay un >>
                                    fichero2 = open(instruccion->salida, O_WRONLY | O_CREAT | O_APPEND, 0600);
                               }else{
                                    fichero2 = open(instruccion->salida, O_WRONLY | O_CREAT | O_TRUNC , 0600);
                               }
```

```
                    if (fichero2 == -1) {
                        fprintf(stderr,"%i: Error. Fallo al abrir el fichero de redirección de salida\n", fichero2);
                        exit(1);
                    } else {
                        dup2(fichero2,1); //redirigimos la salida
                    }
                    execvp(instruccion->comandos[0].nombreComando, instruccion->comandos[0].argv); //ejecutamos el comando
                    fprintf(stderr,"%s: No se encuentra el comando\n",instruccion->comandos[i].nombreComando);
                }else { //si solo hay redirección de entrada
                    execvp(instruccion->comandos[0].nombreComando, instruccion->comandos[0].argv); //ehecutamos el comando
                    fprintf(stderr,"%s: No se encuentra el comando\n",instruccion->comandos[i].nombreComando);
                }
            } else if (instruccion->salida != NULL) { //si solo hay redirección de salida
                int fichero;

                if(instruccion->concatenar == 1){ //si hay un >>
                    fichero = open(instruccion->salida, O_WRONLY | O_CREAT | O_APPEND, 0600);
                }else{
                    fichero = open(instruccion->salida, O_WRONLY | O_CREAT | O_TRUNC , 0600);
                }

                if (fichero == -1) {
                    fprintf(stderr,"%i: Error. Fallo al abrir el fichero de redirección de salida\n", fichero);
                    exit(1);
                } else {
                    dup2(fichero,1); //redirigimos la salida
                }
                execvp(instruccion->comandos[0].nombreComando, instruccion->comandos[0].argv); //ejecutamos el comando
                fprintf(stderr,"%s: No se encuentra el comando\n",instruccion->comandos[i].nombreComando);
            } else { //si no hay redirecciones

                printf("OPERACIÓN:\n");
                execvp(instruccion->comandos[0].nombreComando, instruccion->comandos[0].argv); //ejecutamos el comando
                perror("\nError en exec\n");
                fprintf(stderr,"%s: No se encuentra el comando\n",instruccion->comandos[i].nombreComando);
                exit(-1);
            }

        }else{ //el padre
            while (wait(&status) != pid)
            ; //espera por su hijo
        }

      /*-------------------------------------------------------------------------*/
    /*-----------------------------2 O MAS COMANDOS-----------------------------*/
    }else if(instruccion->nComandos >= 2){
        pidHijos = malloc(instruccion->nComandos * sizeof(int));
        pipes = (int **) malloc ((instruccion->nComandos-1) * sizeof(int *)); //Reservamos memoria para la matriz de pipes

        for(i=0; i<instruccion->nComandos-1; i++){ //Creo tantos pipes como comandos-1
            pipes[i] = (int *) malloc (2*sizeof(int));
            if(pipe(pipes[i]) < 0)
                fprintf(stderr, "Fallo al crear el pipe %s/n" , strerror(errno));
        }
        for(i=0; i < instruccion->nComandos; i++){ //Por cada comando hago un HIJO
            pid = fork();

            if(pid < 0){
                fprintf(stderr, "Fallo el fork() %s\n" , strerror(errno));
                exit(-1);
            } else if(pid == 0){ //Soy el HIJO
                if(i == 0){ //Si soy el primer comando
                    if(instruccion->entrada != NULL){ //Si hay redirección de entrada
                        fichero = open(instruccion->entrada, O_RDONLY);
                        if (fichero == -1) {
                            fprintf(stderr,"%i: Error. Fallo al abrir el fichero de redirección de entrada\n", fichero);
                            exit(1);
                        } else {
                            dup2(fichero,0); //Redirijimos la entrada
                        }
                    }
                    for(j=1; j<instruccion->nComandos-1; j++){ //Por cada pipe creado (menos el que voy a utilizar) cierro su p[1] y p[0]
                        close(pipes[j][1]);
                        close(pipes[j][0]);
                    }
                    close(pipes[0][0]);
                    dup2(pipes[0][1],1);
                }
                if(i>0 && i<instruccion->nComandos-1){ //Si soy un comando intermedio
                    if(i==1 && instruccion->nComandos != 3){ //Si soy el segundo comando
                        for(j=i+1; j<instruccion->nComandos-1; j++){ //Por cada pipe creado (menos el que voy a utilizar) cierro su p[1] y p[0]
                            close(pipes[j][1]);
                            close(pipes[j][0]);
                        }
                    }
                    if(i==instruccion->nComandos-2 && instruccion->nComandos != 3){ //Si soy el penultimo comando
                        for(j=0; j<i-1; j++){ //Por cada pipe creado (menos el que voy a utilizar) cierro su p[1] y p[0]
                            close(pipes[j][1]);
                            close(pipes[j][0]);
                        }
                    }
```

```
309                         if(i!=1 && i!=instruccion->nComandos-2 && instruccion->nComandos != 3){ //Si no soy ni el segundo ni el penultimo
310                             for(j=0; j<i-1; j++){ //Por cada pipe creado (menos el que voy a utilizar) cierro su p[1] y p[0]
311                                 close(pipes[j][1]);
312                                 close(pipes[j][0]);
313                             }
314                             for(j=i+1; j<instruccion->nComandos-1; j++){ //Por cada pipe creado (menos el que voy a utilizar) cierro su p[1] y p[0]
315                                 close(pipes[j][1]);
316                                 close(pipes[j][0]);
317                             }
318                         }
319                         close(pipes[i-1][1]);
320                         dup2(pipes[i-1][0],0);
321                         close(pipes[i][0]);
322                         dup2(pipes[i][1],1);
323                     }
324                     if(i == instruccion->nComandos-1){ //Si soy el ultimo comando
325                         if (instruccion->salida != NULL) { //Si hay redirección de salida
326 
327                             if(instruccion->concatenar == 1){ //si hay un >>
328                                 fichero = open(instruccion->salida, O_WRONLY | O_CREAT | O_APPEND, 0600);
329                             }else{
330                                 fichero = open(instruccion->salida, O_WRONLY | O_CREAT | O_TRUNC , 0600);
331                             }
332                             if (fichero == -1) {
333                                 fprintf(stderr,"%i: Error. Fallo al abrir el fichero de redirección de salida\n", fichero);
334                                 exit(1);
335                             } else {
336                                 dup2(fichero,1); //Redirigimos la salida
337                             }
338                         }
339 
340                         for(j=0; j<instruccion->nComandos-2; j++){ //Por cada pipe creado (menos el que voy a utilizar) cierro su p[1] y p[0]
341                             close(pipes[j][1]);
342                             close(pipes[j][0]);
343                         }
344                         close(pipes[i-1][1]);
345                         dup2(pipes[i-1][0],0);
346                     }
347                     execvp(instruccion->comandos[i].nombreComando, instruccion->comandos[i].argv); //Ejecutamos el comando
348                     fprintf(stderr,"%s: No se encuentra el mandato\n",instruccion->comandos[i].nombreComando);
349                 } else { //Soy el PADRE
350                     pidHijos[i] = pid; //Guardamos el PID del HIJO
351                 }
352             }
353             for(k=0; k<instruccion->nComandos-1; k++){ //Cerramos todos los pipes
354                 close(pipes[k][1]);
355                 close(pipes[k][0]);
356             }
357             for(k=0; k<instruccion->nComandos; k++){
358                 waitpid(pidHijos[k],NULL,0);
359             }
360             for(i=0; i<instruccion->nComandos-1; i++){ //Liberamos memoria
361                 free(pipes[i]);
362             }
363             free(pipes);
364             free(pidHijos);
365         }
366         /*-------------------------------------------------------------------------*/
367 
368     return;
369 }
```

Creación de la estructura *línea*

Primeramente, se inicializarán todas las variables necesarias para la ejecución de dicha función:

- Variables para iterar
- Un entero para almacenar el valor de un hijo y otro para guardar su status, en caso que la línea sólo contenga un comando.
- Un entero para almacenar el valor de retorno al abrir un fichero con la llamada *open()*, ya sea para escritura o lectura.
- Un apuntador a entero para trabajar con dos o más posibles hijos de acuerdo a el número de comandos que se encuentren en la línea.
- Una matriz de pipes para lograr la comunicación entre los procesos hijos, en caso de que se detecten dos o más comandos.
- Un grupo de 5 arreglos de cadenas y un arreglo de 5 estructuras de tipo *comando*. A cada uno le corresponderá un arreglo para su campo *argv*, que almacenará los posibles argumentos de cada comando.
- Por último, la estructura que guardará todo, un apuntador de tipo *línea* al que se le asignará el arreglo de comandos del punto anterior.

Lo primero será obtener el número de comandos encontrados mediante la función *numeroComandos(*tokens*)* y asignando el valor al atributo *nComandos* de la estructura línea ya creada.

Posteriormente se verificará si existe algún redireccionamiento con las funciones *Entrada*(tokens, instrucción) y *Salida*(tokens, instrucción), actualizando los atributos respectivos de la estructura según sea el caso.

Una vez hecho esto, se procederá a almacenar los comandos y sus respectivos argumentos (véase de la línea 148 – 174 del código fuente). Se iterará *nComandos* veces, y por cada comando que se encuentre, se almacenará su nombre y sus argumentos en la estructura *línea*.

Finalmente tenemos completa la estructura de tipo *línea*, la cual guarda toda la información necesaria para la ejecución de los comandos.

Ejecución de los comandos

- **Condición 1:** *Sólo hay un comando y es exit o cd*

    - *Subcondición 1: EXIT*
      Si el primer comando de la estructura *línea* es "exit" se terminará la ejecución del programa.

    - *Subcondición 2: CD*
      Si el primer comando de la estructura *línea* es "cd" se hará una llamada al sistema con *chdir( )*, pasándole como argumento el segundo argumento del primer comando, el cual deberá ser ".." o una dirección válida. Esto cambiará el directorio de trabajo actual.

- **Condición 2:** *Sólo hay un comando y no es exit o cd*
  Se creará un único proceso hijo con *fork( )* el cual ejecutará el comando.

    - *Subcondición 1: Sólo hay direccionamiento de entrada*
      Se hará uso de la llamada a sistema *open( )*, pasándole como primer argumento el atributo *entrada* de la estructura línea y como segundo argumento el modo O_RDONLY para definir que sólo tendrá permiso de lectura, el resultado se guarda en la variable *fichero* y si logró abrir el archivo se hace una llamada con *dup2(*fichero,*0)*, reemplazando el descriptor de archivo 0 (stdin) con el fichero y mandando a ejecutar el comando mediante *execvp( )*, siendo el primer argumento el nombre del comando guardado en el primer comando en su atributo nombreComando y de la misma manera su arreglo de argumentos.

- *Subcondición 2: Sólo hay direccionamiento de salida*

  Se pregunta por la bandera *concatenar* de la estructura, si tiene un valor de '1', la forma de apertura del archivo será de la forma:

  *open(*línea->salida, O_WRONLY | O_CREAT | O_APPEND, 0600*);*

  si no:

  *open(*linea->salida, O_WRONLY | O_CREAT | O_TRUNC, 0600*);*

  Donde:

  O_WRONLY: bandera que define el modo de apertura como sólo escritura.

  O_CREAT: bandera que crea el archivo en caso de que no exista.

  O_APPEND: bandera que permitirá posicionarse al final del archivo.

  O_TRUNC: bandera que elimina el contenido del archivo en caso de que exista.

  0600: permisos para el usuario, lectura y escritura.

  De forma análoga, si el archivo pudo abrirse correctamente, se reemplazará el descriptor de archivo 1 (stdout) con la llamada *dup2(*fichero,1*)* y se ejecutará el comando con *execvp( ).*

- *Subcondición 3: Hay ambos direccionamientos*

  Al identificarse un direccionamiento de entrada, se usará el primer fichero y llevará a cabo los pasos de la subcondición 1, posteriormente, con otro fichero, los pasos de la subcondición 2. Una vez redireccionados la entrada y la salida, se procede nuevamente a hacer la llamada *execvp( ) con el comando y sus argumentos.*

- *Subcondición 4: No hay direccionamientos*

  Si tanto el atributo *entrada* como el de *salida* de la estructura apuntan a NULL, el comando se ejecutará de forma normal con la llamada a *execvp( ).*

NOTA: En todas las subcondiciones el proceso hijo será el encargado de ejecutar el comando, mientras que el proceso padre esperará a que este termine mediante la función *wait( )* y el valor de *status.*

- **Condición 3:** *Hay más de un comando*

  Se asignará memoria para el arreglo de procesos hijos *pidHijos*, teniendo un hijo por cada comando. De la misma manera se asignará memoria para la matriz de pipes de tamaño *(nComandos -1) *2*. Es decir, *nComandos – 1* arreglos de 2 celdas, una para lectura en el pipe y otra para escritura en el pipe.

  Por cada comando se creará un proceso Hijo y existirán distintos casos:

  - *Comando inicial*

    En caso de existir redireccionamiento de entrada, se abrirá el archivo en modo de lectura y mediante la función *dup2(*fichero,0), se reemplazará la entrada estándar con el archivo para este proceso.

    Se cerrarán todos los pipes, tanto su puerto de lectura como de escritura, a excepción del primer pipe que conservará su entrada, con ello, se hará uso de *dup2(*pipe[0][1], 1). Es decir, se reemplaza la salida estándar de este proceso por la escritura del primer pipe.

  - *Comando intermedio*

    - *2do Comando*
      Cierra todos los pipes, a excepción del de esta iteración, cierra la entrada del primer pipe, reemplaza la entrada estándar del proceso actual por la salida del pipe anterior. De la misma manera, cierra su propia salida y reemplaza la salida estándar por la escritura de este pipe.

    - *Penúltimo*
      Cierra todos los pipes anteriores, lectura y escritura a excepción del actual, del cual repite lo anterior, conecta la salida del pipe anterior con la entrada estándar del proceso actual y luego reemplaza la salida estándar con la escritura del pipe actual.

    - *Ni el 2do ni el penúltimo*
      Mismo procedimiento anterior, sólo que se cierran los pipes anteriores y los posteriores al pipe actual. De igual manera se conecta la lectura del pipe anterior a la entrada estándar y se reemplaza la salida estándar con la escritura del pipe actual.

NOTA: Existe otra ligera consideración cuando el número de comandos es 3, puesto que el penúltimo y el segundo comando es el mismo.

- *Comando final*

  En caso de existir redireccionamiento de salida, se abrirá el archivo en modo de escritura, dependiendo si la bandera de *concatenación* esté o no activa se abrirá como O_APPEND o como O_TRUNC y mediante la función *dup2(*fichero,1), se reemplazará la salida estándar con el archivo para este proceso.

  Cierra todos los pipes anteriores y por último reemplaza la entrada estándar con la lectura del pipe anterior.

Sin importar a qué condición entre el comando, se ejecutará la función *execvp( )* con el nombre del comando de la iteración y sus respectivos argumentos.

Por otro lado, el padre guardará los pid en el arreglo previamente creado, para posteriormente esperar a que termine cada uno de sus procesos hijo.

Por último, se cierran todos los pipes y se libera la memoria para todas las variables dinámicas y el proceso vuelve a repetirse hasta que el usuario ingrese “exit”.

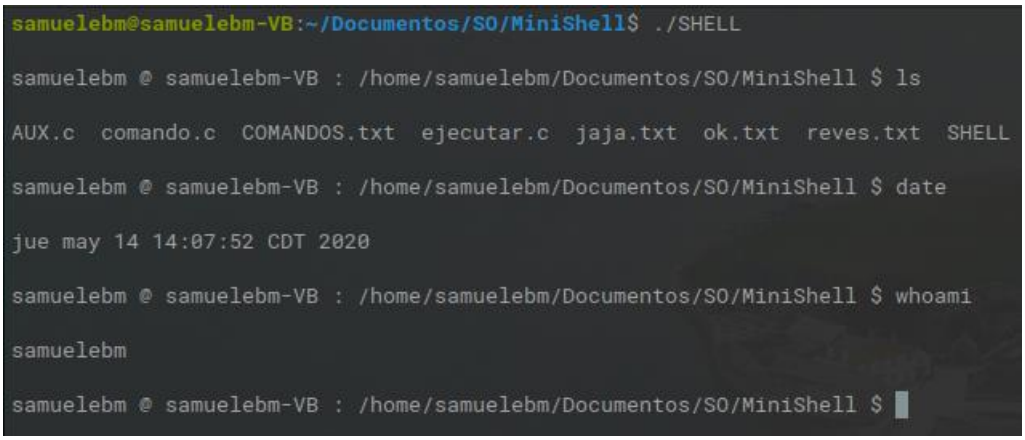
```
void main(void){
     char line[1024];             // COMANDO COMPLETO
     char *argv[64];              // ARGUMENTOS
     char usuario[20], computadora[20], directorio[1024];

     register struct passwd *pw;
     register uid_t uid;
     uid = geteuid();
     pw = getpwuid(uid);

     gethostname(computadora, sizeof(computadora));

     while (1) {
          getcwd(directorio, sizeof(directorio));
          printf("\n%s @ %s : %s $ ",pw->pw_name,computadora,directorio);
          gets(line);               // Lee el input
          printf("\n");
          parse(line, argv);        // Divide la cadena en Tokens
          //imprimeTokens(argv);

          ejecutar(argv);
     }
}
```

Función Main

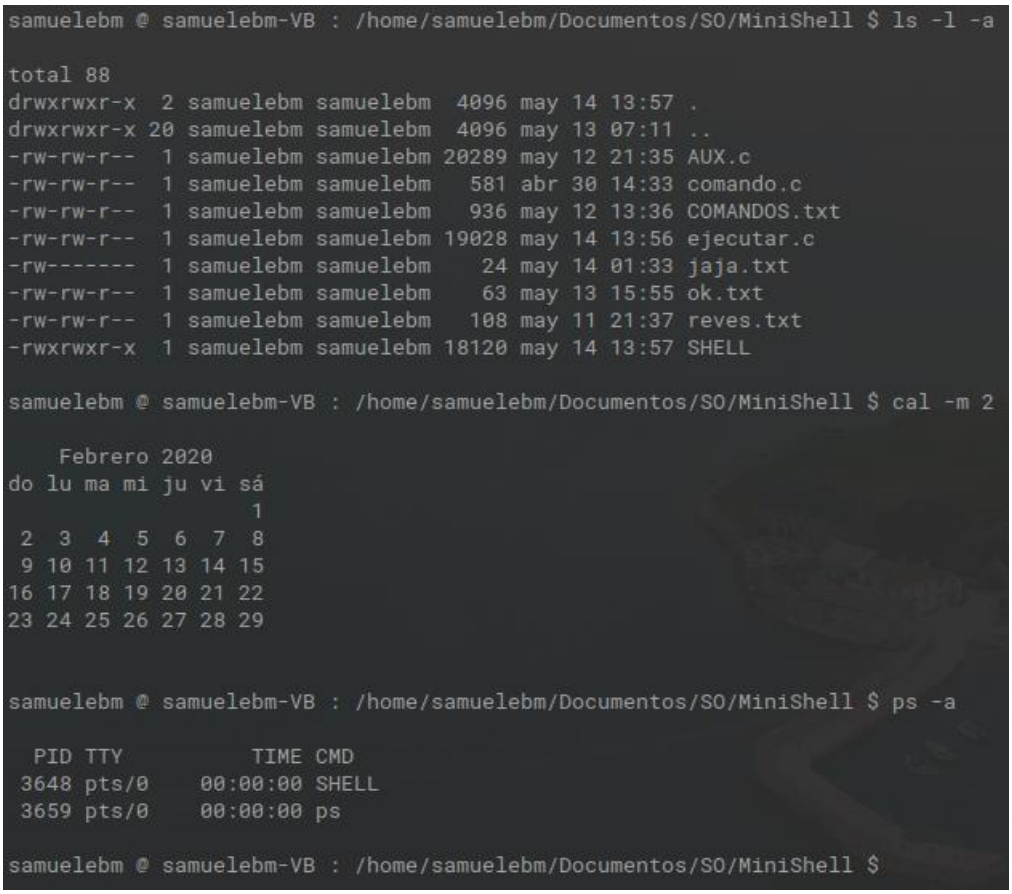
Se obtendrán las variables de sistema mediante arreglos y llamadas al sistema, encontrando así el nombre del usuario actual, el nombre de su computadora y el directorio de trabajo actual. Se imprimirá una línea con esta información y se esperará a que el usuario ingrese una línea de comandos, para obtener sus tokens y llamar a la función *ejecutar( )*, pasando como argumento dichos tokens. Todo esto en un ciclo infinito hasta que el usuario ingrese “exit”.

Resultados

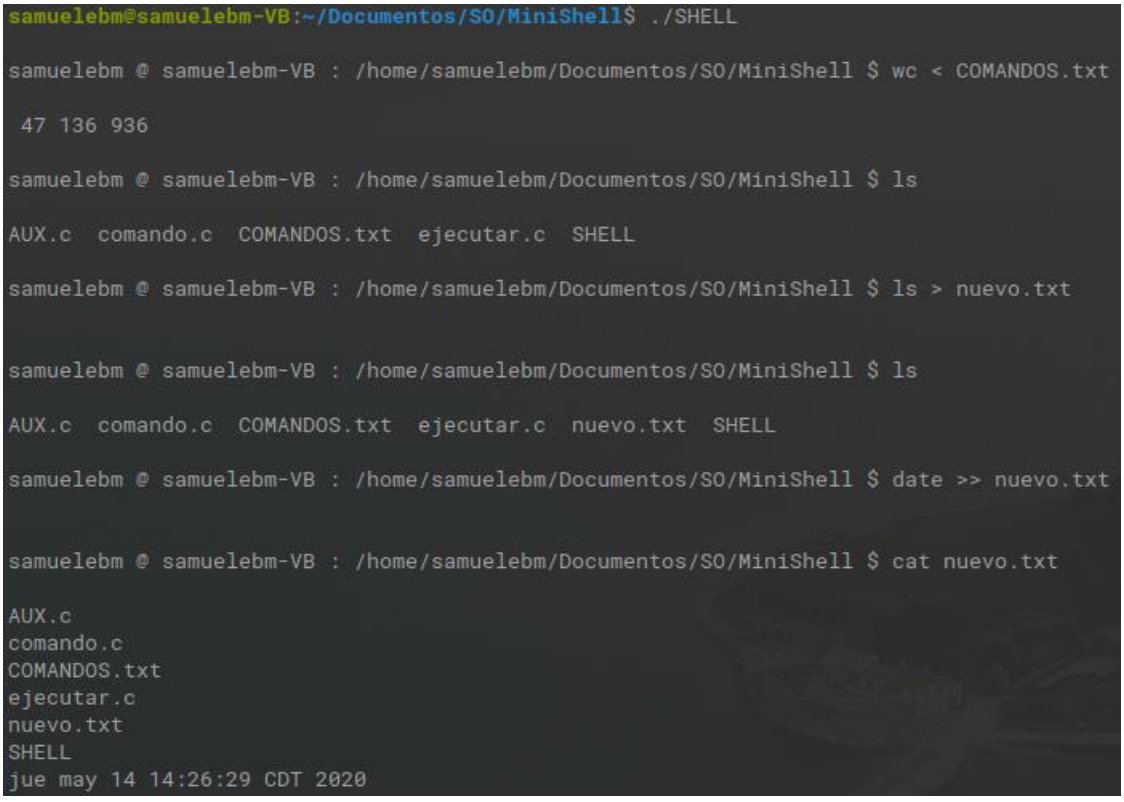
- Comando sin argumentos



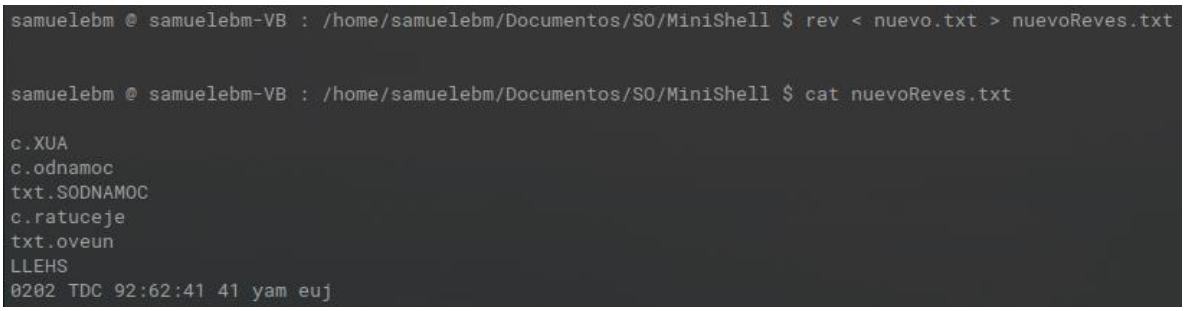
- Comando con argumentos



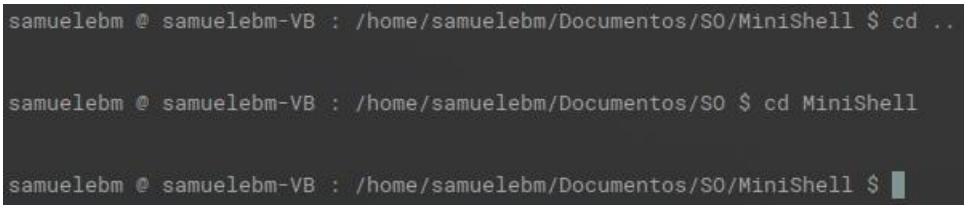
- Comando con redireccionamiento de entrada o de salida



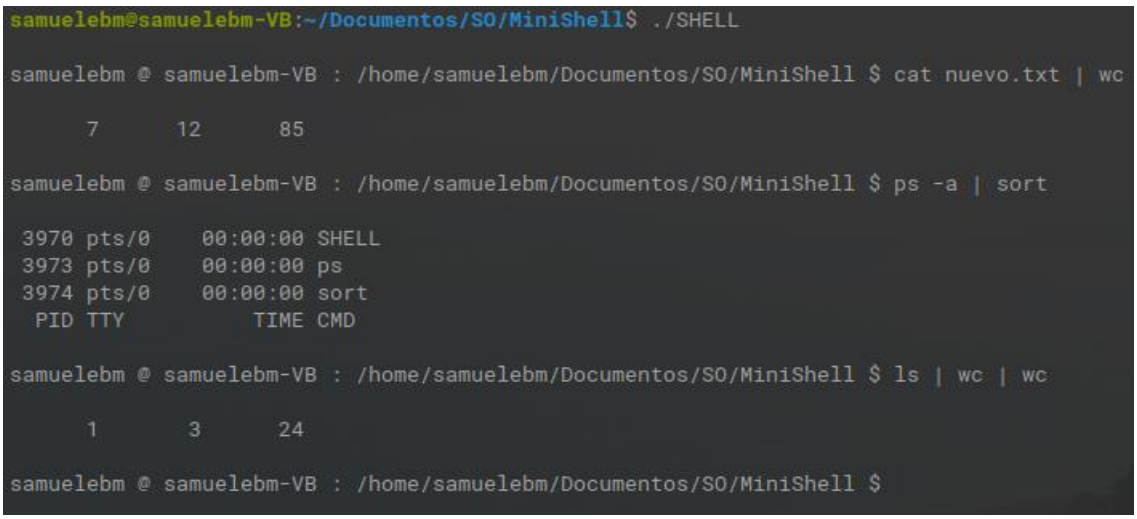
- Comando con ambos direccionamientos



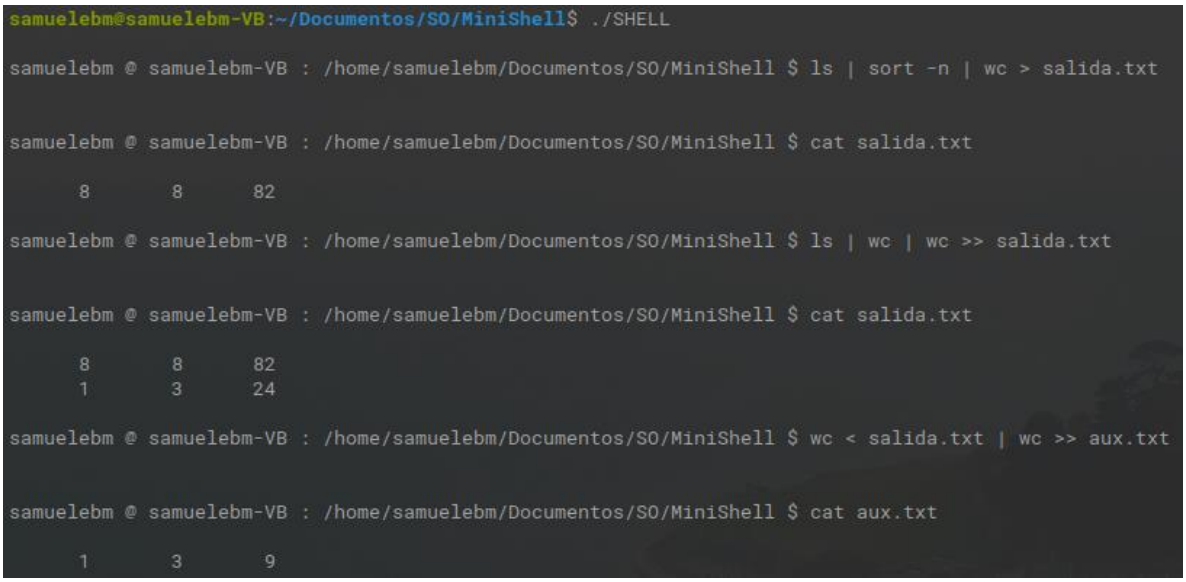
- Comando CD



- Múltiples comandos sin direccionamientos



- Múltiples comandos con direccionamientos



- Exit

Conclusiones

Con la realización de este proyecto, fue posible poner en práctica algunos de los conocimientos adquiridos a lo largo del curso, especialmente del primer y segundo parcial, donde se aprendió la creación, comunicación y ejecución de procesos, mediante llamadas al sistema como lo son: *fork( ), pipe( ), dup( ), exec( ),* entre otras.

Entender el uso y aplicación de cada una de estas funciones ha sido vital para la implementación de este proyecto, por lo cual ha sido muy enriquecedor, ya que se han podido reafirmar todos los conocimientos de la materia y de cómo es que funciona una pequeña parte del sistema operativo.

Fue especialmente un reto lograr la comunicación entre los múltiples procesos cuando existía una línea con múltiples comandos, puesto que aparecieron muchos casos a considerar con las tuberías y los descriptores de archivo de cada proceso hijo, sin embargo, se logró solucionar e incluso optimizar esta sección de código.

Por otro lado, el uso de estructuras facilitó la implementación de la práctica en gran medida, además de modular el problema con diferentes funciones que poco a poco completaban dichas estructuras.

Ha sido sin duda un desafío que resume en gran medida muchos de los conocimientos obtenidos en el curso.

Referencias

Anónimo. (Sin mención). UNIX y LINUX. 13 mayo 2020, de Comav Sitio web: https://bioinf.comav.upv.es/courses/unix/unix_intro.html

Cortés, H. (2011). Shell (informática). 13 mayo 2020, de Blogspot Sitio web: http://ingenieriaensistemasxajogardu.blogspot.com/2011/11/shell-informatica.html

Carlos Villagómez. (2017). Linux - "Shell". 13 mayo 2020, de CCM Sitio web: https://es.ccm.net/contents/316-linux-shell

CyberDisk. (2008). Resumen de Comandos UNIX más Importantes. 13 mayo 2020, de CyberDisk Sitio web: https://www.uv.es/sto/libros/cyberdisk/alice/libro/comunix.htm