



INSTITUTO POLITÉCNICO NACIONAL



ESCUELA SUPERIOR DE CÓMPUTO

SISTEMAS OPERATIVOS

PROYECTO: MINISHELL

CASTRO CRUCES JORGE EDUARDO

CORTÉS CASTAÑO EDUARDO

2CM8

ANA BELEM JUÁREZ MÉNDEZ

Contenido

Objetivo	3
Introducción	4
Desarrollo	5
Solución.....	6
Código	8
Resultados.....	15
Conclusiones	18
Referencias	19

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, ...

Así mismo, se busca que el alumno aplique los conocimientos que previamente se obtuvieron en materias anteriores, pero principalmente en la asignatura de Sistemas Operativos.

Introducción

La idea básica de este MiniShell es:

- Ejecutar comandos básicos desde la consola (ls, which, pwd, ps, wc, pstree, ...).
- Ejecutar comandos básicos con argumentos (cal -m 2, ls -l -a , ...).
- Ejecutar comandos básicos con |, <, >, >> (ls | wc | wc, ls | sort -n | wc > archivo.txt, cal -m 2 >> archivo.txt, ...).
- Mostrar un prompt dinámico que muestre la siguiente información:
 - Usuario del equipo.
 - El nombre del equipo.
 - Pathname del directorio actual.
- Dejara de pedir comandos al usuario introduzca: *exit*.
- Hacer uso de comunicación entre procesos.
- Hacer uso de llamadas al sistema, como:
 - fork().
 - exec().
 - pipe().
 - dup(), etc.

Desarrollo

Tomando como base la práctica 4 (ejercicio a y ejercicio b), modificar sus programas de tal manera que ahora acepte la ejecución de comandos con: |, <, >, >>. Al igual que la práctica 4, el programa deberá dejar de pedir comandos al usuario, cuando introduzca exit.

El minishell a desarrollar deberá poder ejecutar:

- Comandos (ls, which, pwd, ps, wc, pstree, ...)
- Comandos con argumentos (cal -m 2, ls -l -a, ...)
- Comandos con |, <, >, >> (ls | wc | wc, ls | sort -n | wc > archivo.txt, cal -m 2 >> archivo.txt, ...)

El prompt de su programa deberá estar formato por el usuario del equipo, el nombre del equipo y el pathname del directorio actual. También deberá ser dinámico, es decir, deberá adaptarse según el equipo, usuario y directorio donde se ejecute su programa. A

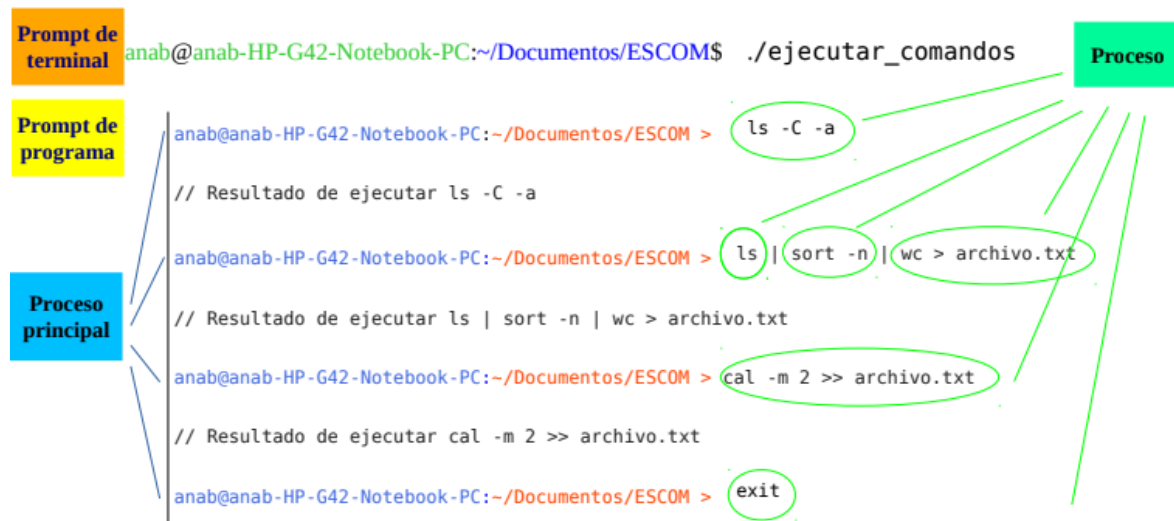
```

anab @ anab-HP-G42-Notebook-PC : home/anab/Documentos/ESCOM $

```

↑ Símbolo de tu elección ↑ Símbolo de tu elección ↑ Símbolo de tu elección
 Usuario del equipo Nombre del equipo Directorio actual

continuación, se muestra un ejemplo:



A continuación, se muestra un ejemplo de cómo se vería su ejecución:

Solución

Struct nodo: Creamos una estructura de datos tipo cola donde almacenamos las cadenas de los comandos a ejecutar.

void Prompt(): Función dedicada a la impresión de el prompt dentro del interprete de comandos, la cual va a recopilar lo que es: El usuario del equipo, el nombre del equipo y el pathname del directorio actual.

void ComandoEntrada(char Comand)*: Función dedicada a abrir y cerrar el fichero al cual el usuario indico que se quiere redireccionar la entrada para algún comando. Recibe como argumento un arreglo de caracteres que contiene el nombre del fichero.

void ComandoSalida(char Comand, int Red)*: Función dedicada a abrir y cerrar el fichero al cual el usuario indico que se quiere redireccionar la salida de algún comando. También, recibe dos argumentos como parámetros, el primero es un arreglo de caracteres que contiene el nombre del fichero y el segundo es una bandera que nos indica si el usuario desea que la salida se agregue hasta el final del fichero o que simplemente se sobrescriba desde el principio.

void CrearProceso(char Argumento[])*: Función dedicada la ejecución de un comando único, esto en el caso de que el usuario solamente ingrese un solo comando a ejecutar. Recibe como argumento un arreglo de caracteres, el cual contiene el comando.

*int analizadorParser(struct nodo **cola)*: Función principal que analiza la cadena introducida y separa mediante uso de tokens los comandos a ejecutar (haciendo uso de la función strtok()). Cuando encuentra un carácter "|", agrega un nuevo nodo a la cola. Cuando se encuentran los caracteres "<", ">" y ">>", se manda a llamar a las funciones ComandoEntrada o ComandoSalida según corresponda.

*int BuscarToken(char *cadena, char *token, char **ap)*: Función que analiza si una subcadena contiene los caracteres de redireccionamiento "<", ">" o ">>", y separa los argumentos de los comandos utilizando la función strtok().

*void Tuberias(struct nodo *cola, int pipeNum):* Ya que se analizó la cadena introducida, se manda a llamar a la función Tuberias(). Se crea un arreglo de tamaño dinámico para manipular la tabla de descriptores de archivo. Después se manda a llamar a la función recursiva CrearProcesoConPipe().

*void CrearProcesoConPipe(struct nodo *cola, int *pipes, int pipeNum, int n):* De manera recursive, se itera la cola creada por analizadorParser, y se ejecutan los comandos introducidos de “izquierda a derecha”, donde la salida del primer comando sirve como entrada para el segundo comando, y así sucesivamente hasta que se llega a un nodo NULL en la cola.

*void liberarMemoria(struct nodo *cola):* Función adicional que limpia el stack de la memoria de las variables alojadas por malloc().

*void Debug(struct nodo *cola, int n):* Función adicional que utilizamos para mostrar los contenidos de la cola en diversas partes del programa (generalmente la mandábamos a llamar junto algún segmento de código donde creíamos que podía haber un problema.

int main(int argc, char argv[]):* Función principal del programa que recibe como argumentos los comandos ingresados por el usuario en la consola.

Código

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include <fcntl.h>
6. #include <sys/types.h>
7. #include <sys/wait.h>
8.
9. #define SALIR 0
10.
11. struct nodo{
12.     char *cadena;
13.     int argc;
14.     char **comandos;
15.
16.     struct nodo *nodo_sig;
17. };
18. //Eduardo Castro
19. void Prompt();
20. void ComandoEntrada(char *Comand);
21. void CrearProceso(char* Argumento[]);
22. void ComandoSalida(char *Comand, int Red);
23. //Eduardo Cortés
24. void Debug(struct nodo *cola, int n);
25. void liberarMemoria(struct nodo *cola);
26. int analizadorParser(struct nodo **cola);
27. void Tuberias(struct nodo *cola, int pipeNum);
28. int BuscarToken(char *cadena, char *token, char **ap);
29. void CrearProcesoConPipe(struct nodo *cola, int *pipes, int pipeNum, int n);
30.
31. int main(int argc, char *argv[]){
32.     struct nodo *cola;
33.
34.     int bandera = 1;
35.     int pipeNum = 0;
36.
37.     int STDOUT=dup(1); //Duplica la salida estandar
38.     int STDIN=dup(0); //Duplica la entrada estandar
39.
40.     while(bandera != SALIR){
41.         int pid, status;
42.
43.         close(1); // Cierro la salida estandar
44.         dup(STDOUT); // Duplico la salida estandar
45.         close(0); // Cierro la entrada estandar
46.         dup(STDIN); // Duplico la entrada estandar
47.
48.         Prompt();
49.         if(pipeNum = analizadorParser(&cola)){
50.             if(pipeNum != -1){
51.                 if(cola->nodo_sig == NULL){
52.                     CrearProceso(cola->comandos);
53.                 }else{
54.                     Tuberias(cola, pipeNum);
55.                 }
56.                 liberarMemoria(cola);
57.             }
58.         }

```



```

59.     bandera = pipeNum;
60. }
61.
62.     return 0;
63. }
64.
65. int analizadorParser(struct nodo **cabeza){
66.     char buffer[256], *aux, dir[256];
67.     int i = 0, adj = 0;
68.     struct nodo *cola;
69.
70.     if(fscanf(stdin, "%[^\n]*c", buffer) && *buffer != EOF){ //Leer del teclado,
        comprobar que el usuario no solo presione Enter.
71.         if(strcmp(buffer, "exit") != 0 && strcmp(buffer, "EXIT") != 0){ //Si el us
            uario introdujo la palabra "exit o EXIT", sale del programa.
72.
73.             *cabeza = (struct nodo*)malloc(sizeof(struct nodo)); //asignar memoria
                para el primer nodo de la cola.
74.             (*cabeza)->nodo_sig = NULL;
75.
76.             cola = *cabeza; //copiar el apuntador a la cabeza de la cola a una var
                iable auxiliar.
77.             /*=====
78.             Primera parte - Tuberías
79.             =====*/
80.             aux = strtok(buffer, "|"); //Separar la cadena introducida donde se en
                cuentra el operador '|'.
81.
82.             while(1){
83.                 cola-
                >cadena = malloc(256*sizeof(char*)); //asignar memoria para la cadena de la tuberí
                a.
84.                 strcpy(cola->cadena, aux); //Copiar el comando.
85.                 aux = strtok(NULL, "|"); //Continuar separando la cadena introduci
                da por el operador '|'.
86.                 if(aux != NULL){
87.                     cola-
                >nodo_sig = (struct nodo*)malloc(sizeof(struct nodo)); //asignar memoria para el s
                iguiente nodo.
88.                     cola = cola-
                >nodo_sig; //Movemos el apuntador al siguiente nodo de la cola.
89.                     cola->nodo_sig = NULL;
90.                 }else{
91.                     break; //Ya no hay más tuberías.
92.                 }
93.             }
94.             //printf("\nPRIMERA PARTE... DONE.\n");
95.
96.             /*=====
97.             Segunda parte - Redirección
98.             =====*/
99.             cola = *cabeza;
100.            while(cola != NULL){ //Recorremos todas las tuberías y buscamos
                operadores de direccionamiento.
101.                strcpy(dir, cola->cadena);
102.                if(BuscarToken(cola->cadena, "<", &aux) != -1){
103.                    ComandoEntrada(aux);
104.                    adj = BuscarToken(dir, ">", &aux);
105.                    if(adj != -1)
106.                        ComandoSalida(aux, adj);
107.                }else{

```

```

108.             adj = BuscarToken(col->cadena, ">", &aux);
109.             if(adj != -1)
110.                 ComandoSalida(aux, adj);
111.             }
112.             cola = cola-
>nodo_sig; //Movemos el apuntador al siguiente nodo de la cola.
113.         }
114.         //printf("\nSEGUNDA PARTE... DONE.\n");
115.
116.         /*=====
117.         Tercera parte - Argumentos
118.         =====*/
119.         cola = *cabeza;
120.         while(col != NULL){ //Recorremos todas las tuberías y separam
os por argumentos.
121.             i = 0; //Inicializamos la variable iteradora.
122.             cola-
>comandos = malloc(256*sizeof(char**)); //asignar memoria para el arreglo de caden
as.
123.             aux = strtok(col->
>cadena, " "); //Separar la cadena del comando en espacios.
124.             while(aux != NULL){
125.                 (cola-
>comandos)[i] = malloc(256*sizeof(char*)); //Asignar memoria para la cadena del ar
gumento.
126.                 strcpy((cola-
>comandos)[i++], aux); //Copiar argumento al arreglo de cadenas.
127.                 aux = strtok(NULL, " "); //Continuar separando la cade
na introducida en espacios.
128.             }
129.             cola->argc = i; //Guardamos el número de argumentos.
130.             cola = cola-
>nodo_sig; //Movemos el apuntador al siguiente nodo de la cola.
131.         }
132.         //printf("\nTERCERA PARTE... DONE.\n");
133.
134.         if(0 == (*cabeza)-
>argc){ //El usuario introdujo únicamente espacios en blanco.
135.             i = 0;
136.         }else{
137.             cola = *cabeza;
138.             i = 0; //Se usará para contar el número de tuberías.
139.             while(col != NULL){
140.                 (cola->comandos)[cola->argc] = NULL;
141.                 cola = cola-
>nodo_sig; //Movemos el apuntador al siguiente nodo de la cola.
142.                 i++;
143.             }
144.         }
145.         }else{
146.             return 0; //El usuario introdujo la palabra "exit o EXIT".
147.         }
148.     }else{ //El usuario no introdujo datos.
149.         i = -1;
150.         getchar(); //Limpiar stdin.
151.     }
152.     return i; //Regresar el número de tuberías.
153. }
154.
155. void Prompt(){ //INICIO DE LA FUNCIÓN Prompt

```

```

156.     int hostname_size=sysconf(_SC_HOST_NAME_MAX); //Funcion que arroja el
    tamaño del hostname
157.     char* whoami=getenv("USER"); //Funcion que arroja el nombre del usuari
    o
158.     char* hostname; //Guarda el nombre del usuario
159.     char cwd[100]; //Guarda la direccion del archivo
160.     hostname=(char*)malloc(hostname_size); //Reserva memoria de forma dina
    mica
161.     gethostname(hostname, hostname_size); //Funcion que retorna el hostnam
    e y lo almacena
162.     getcwd(cwd, sizeof(cwd)); //Funcion que retorna la ruta del archivo y
    lo almacena
163.     printf("%s", whoami); //Imprime el usuario
164.     printf("@%s:", hostname); //Imprime el hostname
165.     printf("~%s$ ", cwd); //Imprime la ruta del archivo
166.     free(hostname);
167. }
168.
169. void ComandoEntrada(char *ComandPtr){
170.     int fd; //Descriptor de fichero
171.
172.     fd=open(ComandPtr, O_RDONLY); //Asigno a la salida el fichero
173.     close(0); //Cierro la entrada estandar
174.     dup(fd); //Duplicamos el descriptor de fichero
175. }
176.
177. void ComandoSalida(char *ComandPtr, int Red){
178.     int Existe=open(ComandPtr, O_RDONLY); //Verifica si el archivo existe
    o no
179.
180.     close(1); //Cierro la salida estandar
181.     if(Red==0){
182.         open(ComandPtr, O_CREAT | O_WRONLY | O_TRUNC, 0777); //Creamos el
    fichero en el que se va almacenar la salida del comando
183.     }else{ //Si el usuario desea que la salida se escriba hasta el final d
    e un archivo que ya existente
184.         if(Existe==0){
185.             open(ComandPtr, O_APPEND | O_WRONLY, 0777); //Abrimos el fiche
    ro y guardamos la informacion hasta el final (El fichero si existe)
186.         }else{
187.             open(ComandPtr, O_CREAT | O_APPEND | O_WRONLY, 0777); //Abrimo
    s el fichero y guardamos la informacion hasta el final (El fichero no existe)
188.         }
189.     }
190. }
191.
192. void CrearProceso(char* Argumento[]){
193.     int status; //Bandera de estado
194.     int pid;
195.     pid=fork(); //Creacion de un proceso hijo el cual ejecutara los comand
    os
196.     if(pid == -1){
197.         perror("ERROR EN LA CREACION DEL PROCESO..."); //Comprobamos si el
    hijo se creó bien
198.     }else if(pid==0){ //Proceso hijo
199.         execvp(Argumento[0], Argumento); //Ejecutamos el comando
200.         perror("MiniShell"); //Error al no poder ejecutar el comando
201.         exit(status); //Finaliza ejecucion del interprete
202.     }else{
203.         pid=wait(&status);
204.     }

```

```

205.     }
206.
207.     void CrearProcesoConPipe(struct nodo *cola, int *pipes, int pipeNum, int n
    ){
208.         if(cola->nodo_sig != NULL){
209.             if (fork() == 0){
210.                 dup2(pipes[(n*2)], 0); // reemplazar el stdin con el read end
de la tubería anterior.
211.                 dup2(pipes[1 + ((n+1)*2)], 1); // reemplazar el stdout con el
write end de la tubería anterior.
212.
213.                 /* se cierran las tuberías */
214.                 for(int i = 0; i < (pipeNum*2); i++){
215.                     close(pipes[i]);
216.                 }
217.
218.                 execvp(*cola->comandos, cola->comandos);
219.             }else{
220.                 CrearProcesoConPipe(cola->nodo_sig, pipes, pipeNum, (n+1));
221.             }
222.         }else{ //Última tubería
223.             if (fork() == 0){
224.                 dup2(pipes[(n*2)], 0); //reemplazamos el stdin con el input re
ad de la tubería anterior
225.
226.                 /* se cierran las tuberías */
227.                 for(int i = 0; i < (pipeNum*2); i++){
228.                     close(pipes[i]);
229.                 }
230.
231.                 execvp(*cola->comandos, cola->comandos);
232.             }
233.         }
234.     }
235.
236.     void Tuberias(struct nodo *cola, int pipeNum){
237.         int status; //Banderas de estado
238.         int pid;
239.         int *pipes;
240.
241.         pipes = malloc(pipeNum*2*sizeof(char*));
242.
243.         /* Inicializar tuberías */
244.         for(int i = 0; i < (pipeNum*2); i++){
245.             pipe(pipes + (i*2)); // Crear las tuberías.
246.         }
247.
248.         // Creamos el fork que ejecutará el primer comando.
249.         if (fork() == 0){
250.             dup2(pipes[1], 1); //reemplazamos el stdout con write de la primer
a tubería.
251.
252.             /* se cierran las tuberías */
253.             for(int i = 0; i < (pipeNum*2); i++){
254.                 close(pipes[i]);
255.             }
256.
257.             execvp(*cola->comandos, cola-
>comandos); //Ejecutar el primer comando.
258.         }else{
259.             CrearProcesoConPipe(cola->nodo_sig, pipes, pipeNum, 0);

```

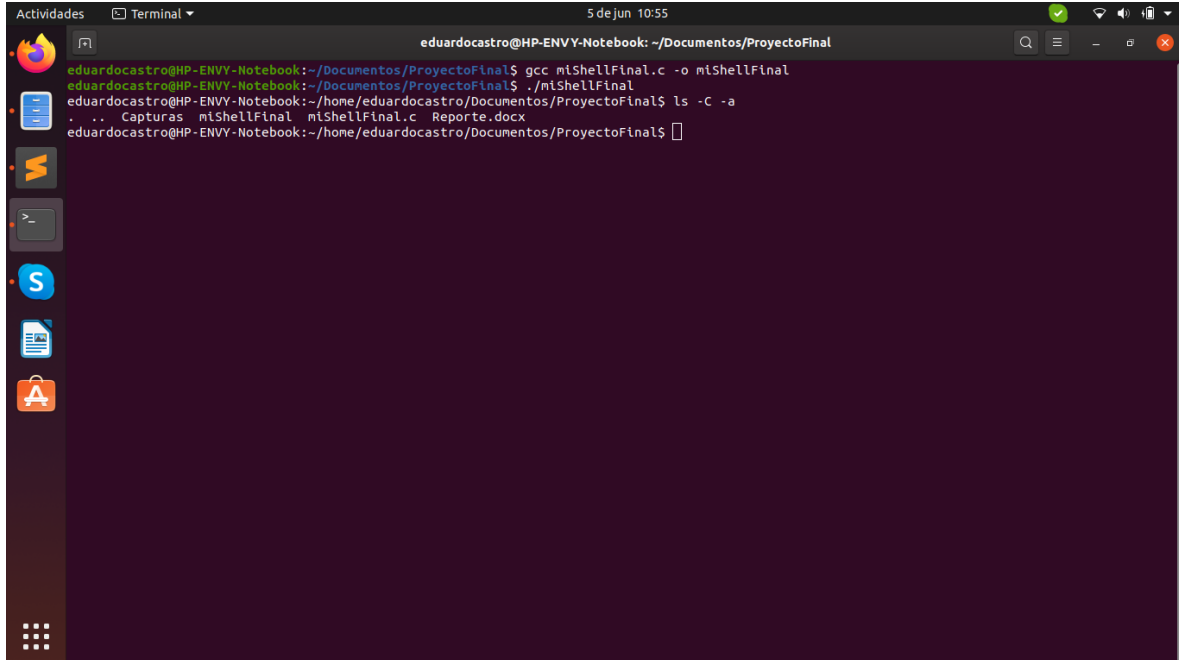
```

260.     }
261.
262.     /* Solo el proceso padre llega aquí; espera a que los hijos terminen */
263.     for(int i = 0; i < (pipeNum*2); i++){
264.         close(pipes[i]);
265.     }
266.     for (int i = 0; i < (pipeNum); i++)
267.         wait(&status);
268.
269.     free(pipes);
270. }
271.
272. int BuscarToken(char *cadena, char *token, char **ap){
273.     char *aux, dir[256], dir2[256], out[256];
274.     int i = 0; //Inicializamos la variable iteradora.
275.     int adj = -1;
276.     while(cadena[i++] != '\0'){
277.         if(cadena[i] == token[0]){
278.             if(cadena[i+1] == token[0])
279.                 adj = 1; // adjuntar al final del archivo.
280.             else
281.                 adj = 0;
282.             aux = strtok(cadena, token); //Separar la cadena del comando p
283.             or el caracter '<'.
284.             while(aux != NULL){
285.                 strcpy(dir, aux);
286.                 aux = strtok(NULL, token); //Continuar separando la cadena
287.                 introducida en espacios.
288.             }
289.             aux = dir;
290.             while(aux[0] == ' ')
291.                 aux++;
292.             strcpy(dir2, aux);
293.             i = 0;
294.             while(dir2[i] != '\0'){
295.                 if(dir2[i] == ' '){
296.                     dir2[i] = '\0';
297.                     aux = dir2;
298.                     break;
299.                 }
300.                 i++;
301.             }
302.             strcpy(out, aux);
303.             *ap = out;
304.             break;
305.         }
306.     }
307.     return adj;
308. }
309.
310. void liberarMemoria(struct nodo *cola){
311.     struct nodo *aux;
312.     while(cola != NULL){
313.         for(int i = 0; i < cola->argc; i++)
314.             free((cola->comandos)[i]);
315.         free(cola->comandos);
316.         free(cola->cadena);
317.         aux = cola;
318.         cola = cola->
319.     }
320.     >nodo_sig; //Movemos el apuntador al siguiente nodo de la cola.

```

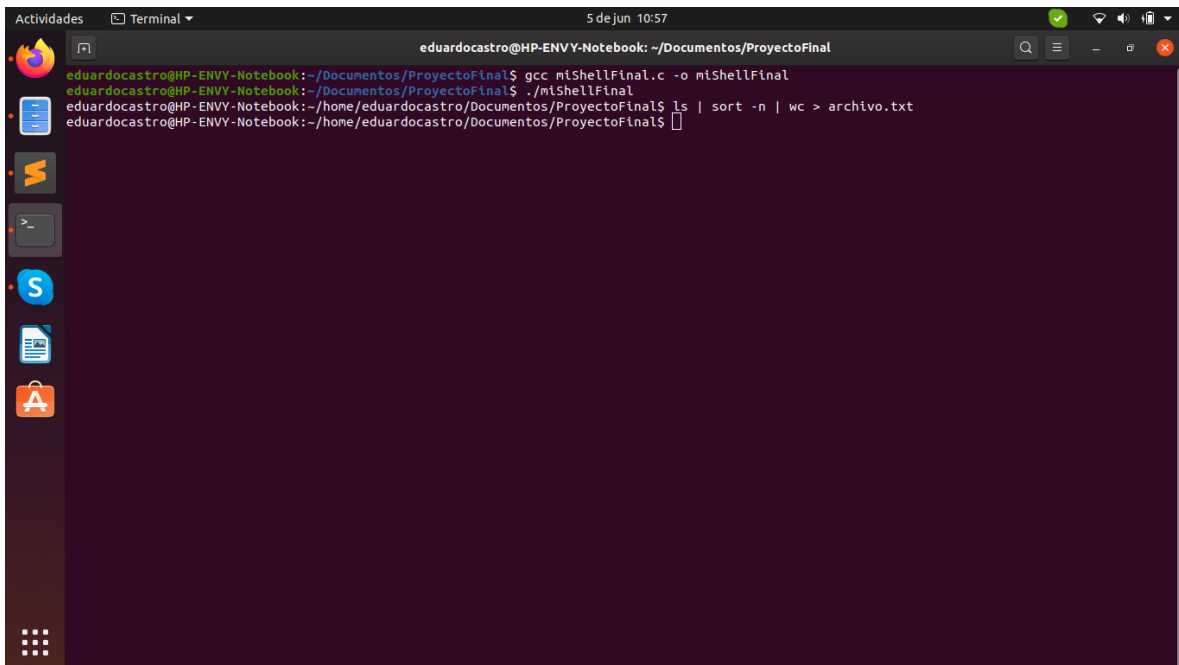
```
317.         free(aux);
318.     }
319. }
320.
321. void Debug(struct nodo *cola, int n){
322.     int pipeI = 0;
323.     printf("\nPipe Num: %d", n);
324.     while(cola != NULL){
325.         printf("\nCMD[%d]: ", pipeI++);
326.         for(int i = 0; i < cola->argc; i++){
327.             printf("\n\tArg[%d]: [%s]", i, (cola->comandos)[i]);
328.         }
329.         cola = cola->
>nodo_sig; //Movemos el apuntador al siguiente nodo de la cola.
330.     }
331.     printf("\n\n");
332. }
```

Resultados



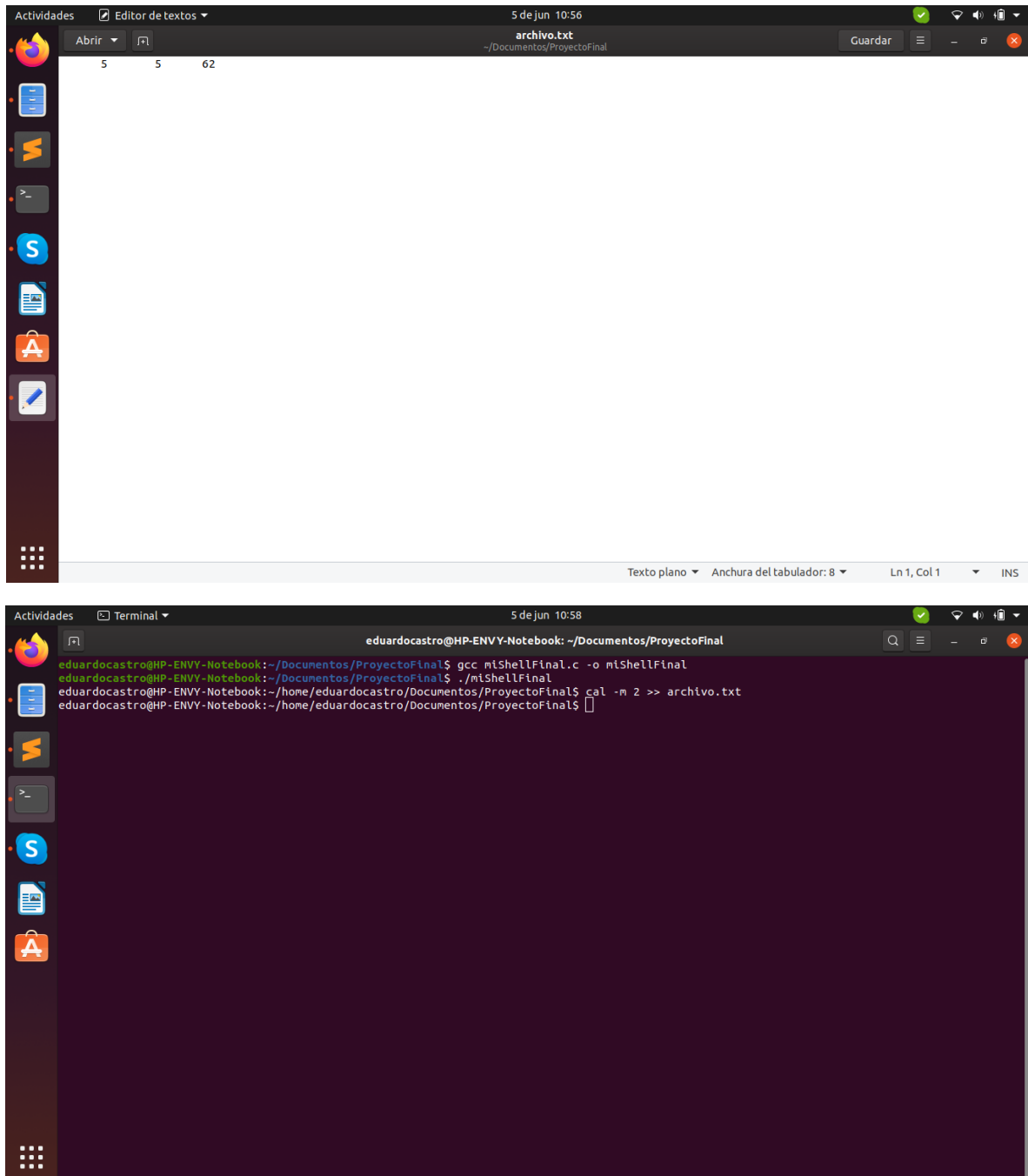
A terminal window titled "Terminal" with the user "eduardocastro@HP-ENVY-Notebook" and the current directory "~/Documentos/ProyectoFinal". The terminal shows the following commands and output:

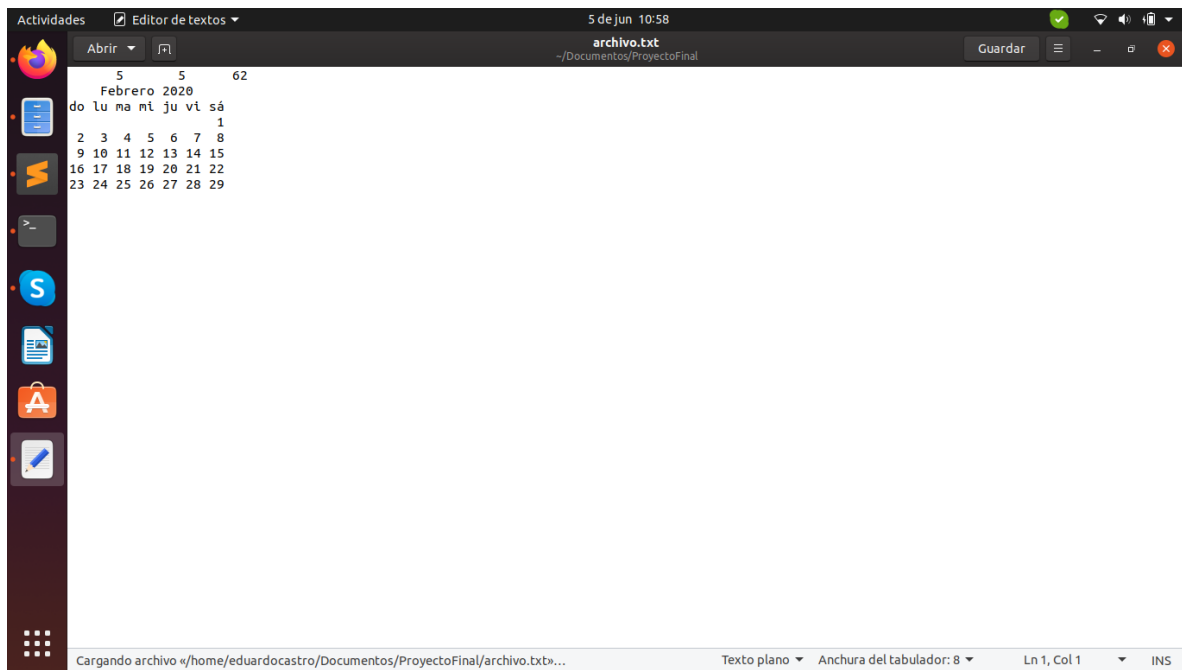
```
eduardocastro@HP-ENVY-Notebook:~/Documentos/ProyectoFinal$ gcc miShellFinal.c -o miShellFinal
eduardocastro@HP-ENVY-Notebook:~/Documentos/ProyectoFinal$ ./miShellFinal
eduardocastro@HP-ENVY-Notebook:~/home/eduardocastro/Documentos/ProyectoFinal$ ls -C -a
. .. Capturas miShellFinal miShellFinal.c Reporte.docx
eduardocastro@HP-ENVY-Notebook:~/home/eduardocastro/Documentos/ProyectoFinal$
```



A terminal window titled "Terminal" with the user "eduardocastro@HP-ENVY-Notebook" and the current directory "~/Documentos/ProyectoFinal". The terminal shows the following commands and output:

```
eduardocastro@HP-ENVY-Notebook:~/Documentos/ProyectoFinal$ gcc miShellFinal.c -o miShellFinal
eduardocastro@HP-ENVY-Notebook:~/Documentos/ProyectoFinal$ ./miShellFinal
eduardocastro@HP-ENVY-Notebook:~/home/eduardocastro/Documentos/ProyectoFinal$ ls | sort -n | wc > archivo.txt
eduardocastro@HP-ENVY-Notebook:~/home/eduardocastro/Documentos/ProyectoFinal$
```





Actividades Editor de textos 5 de jun 10:58

archivo.txt
~/Documentos/ProyectoFinal

Guardar

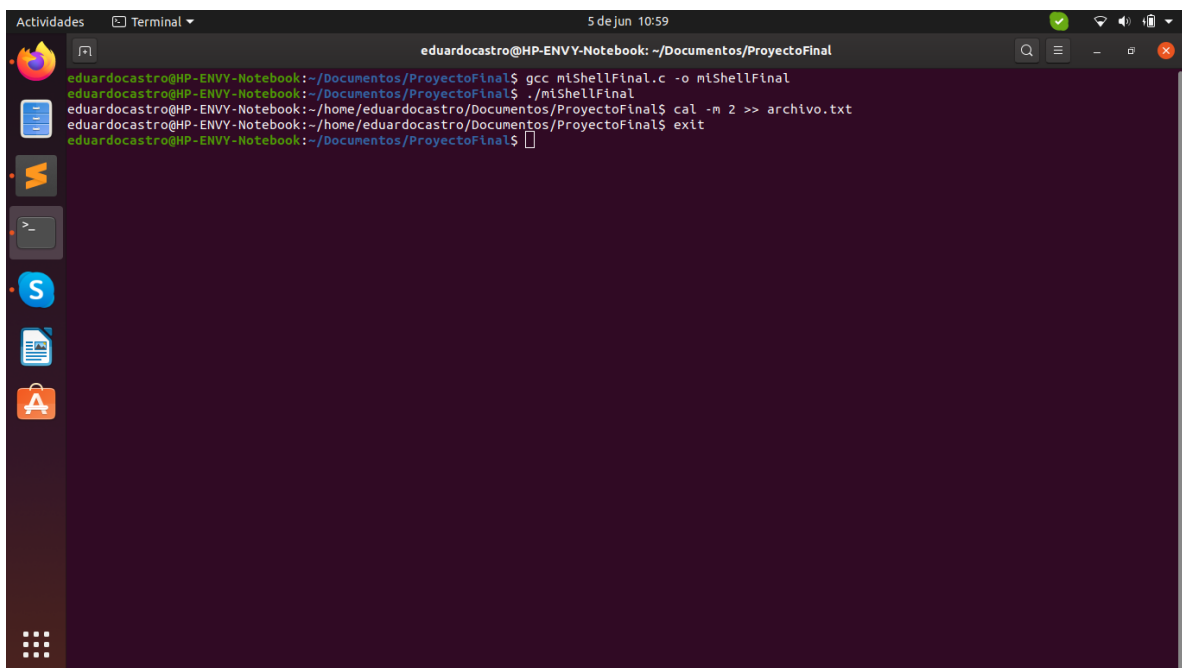
```

5 5 62
Febrero 2020
do lu ma mi ju vi sa
1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29

```

Cargando archivo «/home/eduardocastro/Documentos/ProyectoFinal/archivo.txt»...

Texto plano Anchura del tabulador: 8 Ln 1, Col 1 INS



Actividades Terminal 5 de jun 10:59

eduardocastro@HP-ENVY-Notebook: ~/Documentos/ProyectoFinal

```

eduardocastro@HP-ENVY-Notebook:~/Documentos/ProyectoFinal$ gcc miShellFinal.c -o miShellFinal
eduardocastro@HP-ENVY-Notebook:~/Documentos/ProyectoFinal$ ./miShellFinal
eduardocastro@HP-ENVY-Notebook:~/home/eduardocastro/Documentos/ProyectoFinal$ cal -m 2 >> archivo.txt
eduardocastro@HP-ENVY-Notebook:~/home/eduardocastro/Documentos/ProyectoFinal$ exit
eduardocastro@HP-ENVY-Notebook:~/Documentos/ProyectoFinal$

```

Conclusiones

Finalmente, se cumplieron todos los objetivos propuestos al principio del proyecto. Pudimos realizar un mini intérprete de comandos que admite parámetros, redireccionamiento y tuberías.

Usando las herramientas vistas en este curso y cursos anteriores, utilizamos conceptos como la creación de procesos hijos con el uso de `fork()`, creación y manipulación de tuberías con el uso de `pipe()`, `dup()`, `dup2()` y `close()`, además de retomar conceptos del uso de Estructuras de datos como colas y el manejo y análisis de cadenas.

Con esto, pudimos observar una introducción de las herramientas necesarias para crear un intérprete de comandos.

Por lo cual, damos por concluido satisfactoriamente el proyecto.

Referencias

Ligas principales:

GLENN, James. "Using dup2 for I/O Redirection and Pipes". En <http://www.cs.loyola.edu/~jglenn/702/S2005/Examples/dup2.html>. Consultado el 2020-Junio-01.

CHADWICK, Ryan. "Piping and Redirection". en <https://ryanstutorials.net/linuxtutorial/piping.php>. Consultado el 2020-Junio-01.

"strtok function" en <https://www.cplusplus.com/reference/cstring/strtok/>. Consultado el 2020-Junio-02.

Otras ligas importantes:

<https://man7.org/linux/man-pages/man2/pipe2.2.html>

<https://man7.org/linux/man-pages/man2/open.2.html>

<https://man7.org/linux/man-pages/man2/close.2.html>

<https://man7.org/linux/man-pages/man2/fork.2.html>

<https://baulderasec.wordpress.com/programacion/programacion-con-linux/3-trabajando-con-los-archivos/acceso-de-bajo-nivel-a-archivos/open/>

http://sopa.dis.ulpgc.es/prog_c/FICHER.HTM

http://www.investigacion.frc.utn.edu.ar/labsis/Publicaciones/apunte_linux/mmad.html

<https://pasky.wordpress.com/2009/08/04/funciones-open-close-y-readwrite-en-c/>

<https://www.drivemeca.com/comando-cat-linux/>

<http://mauriciomoo.blogspot.com/2008/03/simbolos-mayor-que-en-ubuntu.html>

<https://www.fing.edu.uy/sysadmin/salas-estudiantiles-linux/comandos-b%C3%A1sicos-de-shell-terminal>

<https://travesuras.wordpress.com/2009/06/01/20090601-1/>

<https://www.por-correo.com/index.php/articulos-de-interes/51-ayuda-linux-listado-de-comandos-importantes-para-linux-unix.html>

https://es.wikipedia.org/wiki/Biblioteca_est%C3%A1ndar_de_C%2B%2B

<https://poesiabinaria.net/2012/03/algoritmos-probar-la-existencia-de-un-fichero-con-open-en-c/>

<https://uniwebsidad.com/libros/python/capitulo-9/metodos-del-objeto-file>

<https://baulderasec.wordpress.com/desde-la-consola/shell-en-unixlinux-sh-ksh-bash/4-bases-de-la-programacion-shell/4-9-los-operadores-logicos-del-shell-y-codigos-de-terminacion/>

<https://www.jabenitez.com/personal/UNI/src/JAShell.c>

<https://www.jabenitez.com/personal/UNI/docu/shell.pdf>

<https://esgeeks.com/comandos-ver-contenido-archivos-linux/>

<https://www.geeksforgeeks.org/bitsttdc-h-c/>