

Instituto Politécnico Nacional Escuela Superior de Cómputo



PROYECTO **Sistemas Operativos.**

“MiniShell”

Integrantes:

*Morales Blas David Israel
Ayona López Eugenio Milton*

Profesora:

Ana Belem Juárez Méndez

Grupo: 2CM8

Fecha de entrega: 1 de junio del 2020

Objetivo

Diseñar y desarrollar, en lenguaje C y en un sistema operativo basado en UNIX, un sistema que funcione como un mini interprete 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, ...

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 practica 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 continuación se muestra un ejemplo:

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 continuación se muestra un ejemplo:

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

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

Introduccion:

Para desarrollar el siguiente proyecto fue necesario aprender y comprender como funcionan los procesos y creación de los mismos con `fork()`, el uso de `dup2()`, `dup()` para poder manipular los descriptores de archivos, `exevp()` para ejecutar los comandos deseados así como entender como se manejan las cadenas y apuntadores de cadenas en el lenguaje C, donde se llevo a cabo el proyecto.

Solución al principal problema de crear la minishell:

1) Saber que librerías se van a usar e importarlas.

```
/*
|   Autor: Eugenio Ayona Milton & Morales Blas David Israel
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pwd.h>
#include <sys/utsname.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <signal.h>

#define MAX_ARGS 20
#define MAX_CHAR 1000
```

- `stdio`: Entrada y salida estándar,
- `string`: Manejo de funciones para manipular cadenas.
- `stdlib`: Cuestion de manejo de memoria dinámica.
- `unistd`: Acceso a la POSIX del sistema operativo (`fork()`, `pipe`, `read`, `write`, `close`, etc)
- `sys/types`: Búsqueda y ordenamiento de directorios y manipulación de archivos.
- `sys/wait`: Esperar procesos.
- `pwd`: Obtener información sobre el sistema donde se encuentra.
- `sys/utsname`: Obtener nombre del equipo.
- `fcntl`: Abrir archivos.
- `sys/stat`: Saber estado de procesos corriendo.
- `signal`: Manipular señales.

2) Dar solución al manejo de cadenas y arreglos de cadenas.

→ Pedir una cadena desde teclado.

```
char *leerLinea(char *comando){  
    int i;  
    gets(comando);  
    return comando;  
}
```

Por medio de gets se pide la cadena, usar fgets no nos servía, no supimos porqué y pese que gets no sea tan seguro se uso para la elaboración del proyecto local.

→ Pasar una cadena a un arreglo de cadenas

```
int extrae_argumentos_general(char *orig, char *delim, char *args[], int max_args, int* IOFlag, int modo, char *cad){  
    char *tmp;  
    int num=0;  
    int band[5], f;  
    char *str = malloc(strlen(orig)+1);  
    //char *cad = malloc(strlen(orig)+1);  
    int flag=0;  
  
    switch (modo)  
    {  
    case 1:  
        //char *str = malloc(strlen(orig)+1);  
        strcpy(str, orig);  
        args[0] = "./comando.out";  
        tmp=strtok(str, delim);  
        do{  
            if (num==max_args){  
                return max_args+1;  
            }  
            args[num]=tmp;  
            num++;  
            tmp=strtok(NULL, delim);  
        }while (tmp!=NULL);  
        return num;  
        break;  
    }
```

Acá solo es necesario mandarle la cadena a la cual se le extraeran las cadenas en este caso “*orig” para que sea un arreglo de cadenas, el delimitador es un espacio “ ”, “char *args[]” contendrá ahora el resultado, los maximos argumentos es para no exceder la memoria, la flag representa un dato que por ahora no se explicará no tiene relevancia en esta función pero si se tocará mas adelante, el modo es para elegir que se haga la operación de extraer cadenas de una cadena (modo 1), “char *cad” ahora no es necesario y no se usa en este paso.

→ Cortar o eliminar un carácter de sobra

```

void removeChar(char *str, char value){
    size_t strl_len = strlen(str);
    size_t i = 0, p=0;
    char result[strl_len];
    for ( i = 0; i < strl_len; i++)
    {
        if (str[i] != value)
        {
            result[p] = str[i];
            p++;
        }
    }
    if(p<strl_len)
        str[p] = '\0';
    for ( i = 0; i < p; i++)
    {
        str[i]=result[i];
    }
}

```

Solo es necesario pasarle una cadena que se almacena en *str, y un carácter almacenado en “value”, se itera sobre la cadena y solo se guarda lo distinto al “value” ya que es lo que queremos eliminar, al final solo se agrega el carácter nulo.

→ Cortar una cadena a determinado rango.

→ Cortar por la Izquierda Es necesario para extraer solo los comandos que se encuentran a la izquierda de determinada flag, podría ser “|, <,>>,<<,>”

```

int extrae_argumentos_leftPipe(char *orig, char *delim, char *args[], int max_args, int* IOFlag){
    char *tmp;
    int band[5], f=1;
    int num=0;
    char *str = malloc(strlen(orig)+1);
    strcpy(str, orig);
    args[0]= "./comando.out";
    tmp=strtok(str, delim);
    do{
        if (num==max_args)
            return max_args+1;
        args[num]=tmp;
        num++;
        tmp=strtok(NULL, delim);
        band[0]=(strcmp(tmp, "|"));
        band[1]=(strcmp(tmp, ">"));
        band[2]=(strcmp(tmp, "<"));
        band[3]=(strcmp(tmp, ">>"));
        band[4]=(strcmp(tmp, "<<"));
        if(band[0]==0){
            f=0;
            *IOFlag=1;
        }
        if(band[1]==0){
            f=0;
            *IOFlag=2;
        }
        if(band[2]==0){
            f=0;
            *IOFlag=3;
        }
        if(band[3]==0){
            f=0;
            *IOFlag=4;
        }
        if(band[4]==0){
            f=0;
            *IOFlag=5;
        }
    }while (f);
    return num;
}

```

Como se observa se toma una cadena, un delimitador y a través de strtok hace el corte, los cambios se guardan en una cadena de cadenas, se le pasa el máximo de argumentos para impedir errores de memoria y una flag donde dependiendo si se encuentran cadenas como “|,<,<<,>,>>” la bandera cambia, cuando se explique la función main se entenderá porqué el uso de una flag.

Para cortar por la derecha se procede a hacer lo inverso de cortar por la izquierda como se observa en la imagen siguiente:

```

int extrae_argumentos_rightPipe(char *orig, char *delim, char *args[], int max_args, int *IOFlag){

    char *tmp;
    int band[5];
    char *str = malloc(strlen(orig)+1);
    int flag=0;
    int num=0;
    strcpy(str, orig);
    args[0]= "./comando.out";
    tmp=strtok(str, delim);
    do{
        band[0]=strcmp(tmp,"|");
        band[1]=strcmp(tmp,">");
        band[2]=strcmp(tmp,"<");
        band[3]=(strcmp(tmp,">>"));
        band[4]=(strcmp(tmp,"<<"));
        if (band[0]==0){
            flag=1;
            *IOFlag=1;
            tmp=strtok(NULL, delim);
        }
        if (band[1]==0){
            flag=1;
            *IOFlag=2;
            tmp=strtok(NULL, delim);
        }
        if (band[2]==0){
            flag=1;
            *IOFlag=3;
            tmp=strtok(NULL, delim);
        }
        if(band[3]==0){
            flag=1;
            *IOFlag=4;
        }
        if(band[4]==0){
            flag=1;
            *IOFlag=5;
        }
        if (flag==1){
            args[num]=tmp;
            num++;
        }

        tmp=strtok(NULL, delim);
    }while (tmp!=NULL);
    return num;
}

```

→ Extraer una cadena de cadenas de un determinado rango.

Ahora es necesario cortar la cadena original para que se vaya entendiendo que se usaran solo funciones como `extrae_args_izquierda` o `derecha` con el fin de ahorrar código, por ello a la cadena que se le extraerán los datos es necesario cortarla para ir recorriendo en el comando ingresado.

```

int extrae_argumentos_M(char *orig, char *delim, char *args){

    char *tmp;
    int num=0;
    char *str = malloc(strlen(orig)+1);

    strcpy(str, orig);
    tmp=strtok(str,delim);
    tmp=strtok(NULL,"\\0");
    strcpy(args,tmp);
    return strlen(args)+1;
}

```

Como se observa solo se pasa una cadena y un delimitador así como la cadena nueva a la que se le asignará la cadena cortada, se le reserva memoria a una cadena en este caso “*str”, y se usa strtok para cortar hasta un delimitador, una vez hecho esto se devuelve la longitud de la cadena para después poder iterar en ella o hacer otras operaciones, de ser necesarias.

Otras funciones necesarias son limpiar cadena como se ve en la imagen siguiente, su descripción es sólo como su nombre lo dice limpia una cadena recorriendo en ella y apuntando a nada.

```

void limpiaCad(char *cad[], int lim){
    int i;
    for(i=0; i<lim; i++){
        cad[i]="";
    }
}

```

3) Identificar las variables de usuario como equipo, dirección donde se encuentra:

```

char *nameUser(){
    char *login;
    struct passwd *pentry;
    //log user verify
    if((login = getlogin()) == NULL){
        perror("getlogin");
        exit(-1);
    }
    //Pass user verify
    if((pentry = getpwnam(login)) == NULL){
        perror("getpwnam");
        exit(-1);
    }
    return pentry->pw_name;
}

```

Se verifica que este logueado el usuario para obtener el nombre y se retorna la cadena.

Nos metemos un poco en el main para ver como se llama a la función y a través de un strcmp se copia la información obtenida además se observa como por getcwd se obtiene la dirección.

```

uname(&unameD);
getcwd(dir,sizeof(dir));
strcpy(nombre,nameUser());

```


4) Identificar lo que se necesita programar, es decir : “ | “ , “>,>>,<,<<”

→ Primero identificar esos argumentos, con ello se puede establecer una prioridad para poder resolverlos

verifyCad(), resolverOl() son las funciones encargadas de hacer ello, ahora se explicará su funcionamiento

verifyCad():

```
int verifyCad(char **comand, int tam){
    int priority=0;

    for(int i=0; i<tam; i++){
        if(strcmp(comand[i], "|") == 0 || (strcmp(comand[i], ">") == 0) || (strcmp(comand[i], "<") == 0) || strcmp(comand[i], ">>") == 0 || strcmp(comand[i], "<<") == 0){
            priority++;
        }
    }
    return priority;
}
```

Acá como se puede apreciar por medio de una cadena de caracteres y su tamaño de la misma se itera en ella comparando si existen pipes o redirecciones de entrada y salida, si hay solo se sube la prioridad que estaba inicializada en 0.

resolveOL()

```
int resolveOl(char *orig, char*delim, char*args[], int max_args){
    char *tmp, f=1, *tmp2;
    int num=0, band[5];

    char *str=malloc(strlen(orig)+1);

    strcpy(str, orig);
    args[0]= "./comando.out";
    tmp=strtok(str, delim);
    do{

        band[0]=(strcmp(tmp, "|"));
        band[1]=(strcmp(tmp, ">"));
        band[2]=(strcmp(tmp, "<"));

        band[3]=(strcmp(tmp, ">>"));
        band[4]=(strcmp(tmp, "<<"));
        if(band[0]==0){
            args[num]=tmp;
            num++;
        }
        if(band[1]==0){
            args[num]=tmp;
            num++;
        }
        if(band[2]==0){
            args[num]=tmp;
            num++;
        }
        if(band[3]==0){
            args[num]=tmp;
            num++;
        }
        if(band[4]==0){
            args[num]=tmp;
            num++;
        }
        //tmp2=strtok(NULL, delim);
        tmp=strtok(NULL, delim);

    }while (tmp!=NULL);
    return num;
}
```

resolveOl retorna entonces una cadena de caracteres que sólo esta compuesta por pipes, redireccionamientos de entrada y salida con el fin de saber el orden de los mismos en el comando, de argumentos recibe el comando, un delimitador dado por un espacio cuando se invoca la función, “*args[]” que será donde se guardará la nueva cadena y los max_args cantidad para la cadena máxima. Se usa num para saber al final el tamaño ya que es lo que se retorna.

5) Programar en funciones cada parte es decir una para pipes, otra para redireccion de salida y entrada.

→ Para ejecutar un comando:

```
void oneARG(char *comando){  
  
    int nargs=0 ,temp=0;  
    char *args[0];  
    int flag;  
  
    nargs = extrae_argumentos_general(comando, " ", args, MAX_ARGS,NULL,1,NULL);  
    //priority=verifyCad(&args[0], nargs);  
    sleep(1);  
    if(strcmp(comando,"exit")!=0){  
        args[nargs]=(char *)0;  
        flag = execvp(args[0], args);  
        if(flag == -1){  
            printf("\n [-] There is not found that command or does not exist\n");  
        }  
        else{limpiaCad(&args[0],nargs);  
        }  
    }  
}
```

Sólo necesita el comando y este se verifica que no sea la cadena “exit” de no ser como se sabe execvp() necesita un arreglo terminado en nulo, este arreglo se toma a partir de extrae_argumentos_general que ya se explicó anteriormente, por ello primero a “args[nargs]” se le hace la asignación a (char*)0, y luego sólo se manda a ejecutar el comando, se toma una flag para saber si termino con éxito o no.

→ Para ejecutar un comando que tiene una pipe:

```

void onePipe(char *comando, int *IOFlag){

    int flagIO=*IOFlag, temp=0,fd[2],pid3,nargsLP=0,nargsDP=0,nargsRP=0;
    char comando2[MAX_CHAR],*argsLeft[temp], *argsRight[temp];

    strcpy(comando2,comando);
    nargsLP= extrae_argumentos_leftPipe(comando, " ", argsLeft, MAX_ARGS, &flagIO);
    nargsRP= extrae_argumentos_rightPipe(comando2, " ", argsRight, MAX_ARGS, &flagIO);
    //printf("\n Flag %i", flagIO);
    int status1;
    pipe(fd);
    pid_t p1,p2;
    p1=fork();
    if(p1==0){
        //printf("\r");
        argsRight[nargsRP]= (char *)0;
        close(STDIN_FILENO);
        dup(fd[0]);
        close(fd[1]);
        execvp(argsRight[0], argsRight);
        limpiaCad(&argsLeft[0],nargsRP);
    }if(p1!=0){
        p2=fork();
        if(p2==0){
            argsLeft[nargsLP]= (char *)0;
            close(STDOUT_FILENO);
            dup(fd[1]);
            close(fd[0]);
            execvp(argsLeft[0], argsLeft);
            limpiaCad(&argsLeft[0],nargsLP);
        }
    }
    close(fd[0]);
    close(fd[1]);
    sleep(1);
    fflush(stdin);
    fflush(stdout);
    wait(&p1);
    wait(&p2);
}

```

Como conlleva una “|” implica que son dos comandos a ejecutar, se toma la cadena original sin hacerle cambios guardada en *comando, y la flag como entrada a esta función, es necesario obtener los argumentos de izquierda y derecha del comando para ello se llaman a las funciones extrae_argumentos_leftPipe y rightPipe, después se procesa a hacer un proceso el cual a partir de usar dup para remitir la salida estándar como entrada estándar del proceso a ejecutar ya que son dos y son necesarios dos execvp, se procesa a abrir una tubería y los execvp para ejecutar cada comando, uno tomara la salida del otro en este caso es argsRight el cual al ejecutarse primero se duplica la salida estándar para pasárselo como entrada estándar a este comando.

Al final solo se procede a cerrar las tuberías, y esperar a que terminen los procesos.

→ Redirigir salida a un archivo:

```
void redirectOutputToFile(char *comando,int *IOFlag){

    int flagIO=*IOFlag, temp=0,fd,nargsLP=0,nargsDP=0;
    char comando2[MAX_CHAR],comando3[MAX_CHAR], comando4[MAX_CHAR],*argsLeft[temp];
    pid_t pid3;
    strcpy(comando2,comando);
    strcpy(comando3,comando2);
    //printf("comando : %s ", comando2 );
    nargsLP= extrae_argumentos_leftPipe(comando, " ", argsLeft, MAX_ARGS, &flagIO);
    argsLeft[nargsLP]= (char *)0;
    nargsDP= extrae_argumentos_M(comando2,">",comando3);
    removeChar(comando3,' ');
    pid3=fork();
    //printf("Archivo a re escribir:%s, flag: %i", comando3, flagIO);
    if (pid3 == 0){
        int fd = open(comando3, O_RDONLY | O_WRONLY | O_TRUNC);
        dup2(fd, 1);
        execvp(argsLeft[0], argsLeft);
        close(fd);
        exit(0);
    }
    else{
        wait(&pid3);
    }

    strcpy(comando3," ");strcpy(comando2," ");strcpy(comando4," ");

}
```

Acá es parecido al anterior función pero al hacer el dup se hace con dup2, el cual se lo hace a un archivo para duplicar la salida estandar para pasarselo al archivo como entrada, acá se espera solo un comando por eso solo se extrae un argumento en este caso el izquierdo, y se procede a guardar lo restante como una cadena la cuál será el nombre del archivo.

→ Concatenar salida a un archivo.

```

void concatredirectOutputToFile(char *comando,int *IOFlag){

    int flagIO=*IOFlag, temp=0,fd;
    pid_t pid3;
    char comando2[MAX_CHAR],comando3[MAX_CHAR], comando4[MAX_CHAR],*argsLeft[temp];
    int nargsLP=0,nargsDP=0;
    strcpy(comando2,comando);
    strcpy(comando3,comando2);
    nargsLP= extrae_argumentos_leftPipe(comando, " ", argsLeft, MAX_ARGS, &flagIO);
    argsLeft[nargsLP]= (char *)0;
    nargsDP= extrae_argumentos_M(comando2,">",comando3);
    removeChar(comando3,' ');
    removeChar(comando3,'>');
    pid3=fork();
    //printf("\nArchivo a re escribir:%s, flag: %i\n", comando3, flagIO);
    if (pid3 == 0){
        int fd = open(comando3,O_RDWR| O_APPEND | O_CREAT, S_IRUSR | S_IWUSR | S_IWOTH);
        dup2(fd, 1);
        execvp(argsLeft[0], argsLeft);
        close(fd);
        exit(0);
    }
    else{
        wait(&pid3);
    }
    strcpy(comando," ");
    strcpy(comando3," ");
    strcpy(comando2," ");
    strcpy(comando4," ");
}

```

Es lo mismo que el anterior a excepción de que los permisos al abrir el archivo en `fd = open` cambian ya que ahora se agrega el `O_APPEND` capaz de permitir anexar información sin perder la ya tenida.

→ Abrir la salida de un archivo como entrada estandar, es decir “<” y “<<” se hace como se muestra a continuación:

```

void redirectInputOfFile(char *comando,int *IOFlag){
    int flagIO=*IOFlag, temp=0,fd;
    pid_t pid3;
    char comando2[MAX_CHAR],comando3[MAX_CHAR], comando4[MAX_CHAR],*argsLeft[temp];
    int nargsLP=0,nargsDP=0;
    strcpy(comando2,comando);
    strcpy(comando3,comando2);
    //printf("comando : %s ", comando2 );
    nargsLP= extrae_argumentos_leftPipe(comando, " ", argsLeft, MAX_ARGS, &flagIO);
    argsLeft[nargsLP]= (char *)0;
    nargsDP= extrae_argumentos_M(comando2,"<",comando3);
    removeChar(comando3,' ');
    pid3=fork();
    //printf("Archivo a re escribir:%s, flag: %i", comando3, flagIO);
    if (pid3 == 0){
        int fd = open(comando3,O_RDWR , S_IRUSR);
        close(0);
        dup2(fd, 0);
        close(fd);
        execvp(argsLeft[0], argsLeft);
        exit(0);
    }
    else{
        wait(&pid3);
    }
    strcpy(comando3, " ");
    strcpy(comando2, " ");
    strcpy(comando4, " ");
}

```

Es muy parecido a cuando se redireccionaba la salida estandar pero acá solo se cambia dup2(fd,0) ya que ahora se busca leer la entrada y cambiarla por la salida, así cuando se ejecute execvp tomara la salida de open que se encuentra abierto como como una entrada, se hace lo mismo para concatenar pero sólo cambian los permisos como se muestra en la imagen siguiente:

```

}
void concatredirectInputOfFile(char *comando,int *IOFlag){
    int flagIO=*IOFlag, temp=0,fd;
    pid_t pid3;
    char comando2[MAX_CHAR],comando3[MAX_CHAR], comando4[MAX_CHAR],*argsLeft[temp];
    int nargsLP=0,nargsDP=0;
    strcpy(comando2,comando);
    strcpy(comando3,comando2);
    //printf("comando : %s ", comando2 );
    nargsLP= extrae_argumentos_leftPipe(comando, " ", argsLeft, MAX_ARGS, &flagIO);
    argsLeft[nargsLP]= (char *)0;
    nargsDP= extrae_argumentos_M(comando2,"<",comando3);
    removeChar(comando3,' ');
    removeChar(comando3,'<');
    pid3=fork();
    //printf("Archivo a re escribir:%s, flag: %i", comando3, flagIO);
    if (pid3 == 0){
        int fd = open(comando3,O_RDWR , S_IRUSR);
        close(0);
        dup2(fd, 0);
        close(fd);
        execvp(argsLeft[0], argsLeft);
        exit(0);
    }
    else{
        wait(&pid3);
    }
    strcpy(comando3," ");
    strcpy(comando2," ");
    strcpy(comando4," ");
}

```


Se procedio a desarrollar una función para dos pipes para entender el funcionamiento de 3 comandos con 2 pipes y se explicará a continuación:

```
void twoPipes(char *comando, int *IOFlag){
    //READ_END 0//WRITE END 1
    //CASO PARA DOS PIPES | | EJ LS -LA | GREP U | WC

    int flagIO=*IOFlag, temp=0,fd,fd1[2], fd2[2],status2,nargsLP=0,nargsDP=0, nargsMP=0, nargsRP=0;
    char comando2[MAX_CHAR],comando3[MAX_CHAR], comando4[MAX_CHAR],*argsRight[temp],*argsLeft[temp],*argsCenter[temp];
    pid_t pidC2,pid3;

    strcpy(comando2,comando);
    nargsLP= extrae_argumentos_leftPipe(comando, " ", argsLeft, MAX_ARGS, &flagIO);
    nargsDP= extrae_argumentos_M(comando2,"|", comando3);
    strcpy(comando4,comando3);
    nargsMP=extrae_argumentos_leftPipe(comando3," ", argsCenter, MAX_ARGS, &flagIO);
    nargsRP=extrae_argumentos_rightPipe(comando4," ", argsRight, MAX_ARGS, &flagIO);
    pipe(fd1);
    pidC2 = fork();
    if(pidC2==0){
        close(fd1[0]);
        dup2(fd1[1], STDOUT_FILENO);
        close(fd1[1]);
        argsLeft[nargsLP]= (char *)0;
        execvp(argsLeft[0],argsLeft);
    }else{
        close(fd1[1]);
        pipe(fd2);
        pidC2=fork();
        if (pidC2 == 0){
            close(fd2[0]);
            dup2(fd1[0],STDIN_FILENO);
            close(fd1[0]);
            dup2(fd2[1], STDOUT_FILENO);
            close(fd2[1]);
            argsCenter[nargsMP] = (char *)0;
            execvp(argsCenter[0], argsCenter);
        }else{
            close(fd1[0]);
            close(fd2[1]);
            pid3 = fork();
            if(pid3==0){
                dup2(fd2[0], STDIN_FILENO);
                close(fd2[0]);
                argsRight[nargsRP]=(char *)0;
                execvp(argsRight[0], argsRight);
            }
        }
    }
    wait(&pidC2);
    wait(&pid3);
    wait(&status2);
}
```

El procedimiento es similar a una pipe sólo que acá por ser dos se abren dos procesos (hijos) cada uno se hara cargo de ejecutar una pipe, y se pasaran la salida por medio de tuberias haciendo el duplicado en el descriptor de archivos con dup2, ya se explicó anteriormente para una pipe y el proceso es similar.

Una vez teneindo las operaciones basicas es decir “|, | | .>,<,>>,<<” se procedio a impmentar funciones que usaran estás funciones para simular comandos complejos como ls | wc | wc > ejemplo1.txt

La función twoPipesRedirectOutputToFile sólo recibe el comando y la flag, y bueno ahora al saber que tendra dos pipes y mandar salida a un archivo solo se hace un fork para un hijo capaz de redireccionar la salida de la funcion de twoPipes para el archivo que se desea redirigir la salida como se muestra a continuación, se hace lo mismo para concatenar sólo se cambian los permisos.

```
void twoPipesRedirectOutputToFile(char *comando, int *IOFlag){
    int nargsRP,temp=0, flagIO=*IOFlag;
    char *argsRight[temp];
    char *str = malloc(strlen(comando)+1);
    char *tmp;
    char comando2[MAX_CHAR];
    pid_t pidCASE3;
    strcpy(comando2,comando);
    strcpy(str,comando);
    tmp=strtok(str,">");
    nargsRP=extrae_argumentos_M(comando,">",comando2);
    removeChar(comando2,' ');
    pidCASE3=fork();
    if (pidCASE3 == 0){
        int fd = open(comando2, O_RDONLY | O_WRONLY | O_TRUNC);
        dup2(fd, 1);
        twoPipes(str,&flagIO);
        close(fd);
        exit(0);
    }
    else{
        wait(&pidCASE3);
    }
}

void twoPipesConcatredirectOutputToFile(char *comando, int *IOFlag){
    int nargsRP,temp=0, flagIO=*IOFlag;
    char *argsRight[temp];
    char *str = malloc(strlen(comando)+1);
    char *tmp;
    char comando2[MAX_CHAR];
    pid_t pidCASE3;
    strcpy(comando2,comando);
    strcpy(str,comando);
    tmp=strtok(str,">");
    nargsRP=extrae_argumentos_M(comando,">",comando2);
    removeChar(comando2,' ');
    removeChar(comando2,'>');
    pidCASE3=fork();
    if (pidCASE3 == 0){
        int fd = open(comando2,O_RDWR| O_APPEND | O_CREAT, S_IRUSR | S_IWUSR | S_IWOTH);
        dup2(fd, 1);
        twoPipes(str,&flagIO);
        close(fd);
        exit(0);
    }
    else{
        wait(&pidCASE3);
    }
}
```

Se procede a hacer lo mismo cuando solo tiene una pipe y una redireccion de salida o concatenación de la misma, pero ahora se manda a llamar a onePipe.

```

}
void onePipeconcatredirectOutputToFile(char *comando, int *IOFlag){

    int nargsRP,temp=0, flagIO=*IOFlag;
    char *argsRight[temp];
    char *str = malloc(strlen(comando)+1);
    char *tmp;
    char comando2[MAX_CHAR];
    pid_t pidCASE3;
    strcpy(comando2,comando);
    strcpy(str,comando);
    tmp=strtok(str,">");
    nargsRP=extrae_argumentos_M(comando,">",comando2);
    removeChar(comando2,' ');
    removeChar(comando2,'>');
    pidCASE3=fork();
    if (pidCASE3 == 0){
        int fd = open(comando2,O_RDWR| O_APPEND | O_CREAT, S_IRUSR | S_IWUSR | S_IWOTH);
        dup2(fd, 1);
        onePipe(str,&flagIO);
        close(fd);
        exit(0);
    }
    else{
        wait(&pidCASE3);
    }
}

void onePipeRedirectOutputToFile(char *comando, int *IOFlag){

    int nargsRP,temp=0, flagIO=*IOFlag;
    char *argsRight[temp];
    char *str = malloc(strlen(comando)+1);
    char *tmp;
    char comando2[MAX_CHAR];
    pid_t pidCASE3;
    strcpy(comando2,comando);
    strcpy(str,comando);
    tmp=strtok(str,">");
    nargsRP=extrae_argumentos_M(comando,">",comando2);
    removeChar(comando2,' ');
    pidCASE3=fork();
    if (pidCASE3 == 0){
        int fd = open(comando2, O_RDONLY | O_WRONLY | O_TRUNC);
        dup2(fd, 1);
        onePipe(str,&flagIO);
        close(fd);
        exit(0);
    }
    else{
        wait(&pidCASE3);
    }
}
}

```

6) Unir todo el trabajo para poder realizar comandos complejos como “ ls -la | sort -n | wc >> archivo.txt”


```

int main(int argc, char *argv[]){

    struct utsname uname0;
    int exitt,temp=0, nargs, tuberia[2],nargsLP,priority=0, flagIO;
    char *args[temp], comando[MAX_CHAR], nombre[20], dir[1024], *argsLeft[temp];
    pid_t pid;
    char comando2[MAX_CHAR];
    int verify=0;
    char *args2[temp];
    uname(&uname0);
    getcwd(dir,sizeof(dir));
    strcpy(nombre,nameUser());
    do{
        if(pipe(tuberia)== -1){
            perror("\n [-] ERROR PIPE \n");
            exit(-1);
        }
        //pid_t
        pid=fork();
        if( pid == -1){
            perror("\n [-] ERROR FORK\n");
        }
        else if(pid==0){
            close(tuberia[1]);
            read(tuberia[0],comando, sizeof(comando));
            nargs=0;
            strcpy(comando2,comando);
            verify=resolve01(comando2," ",args2,MAX_ARGS);
            nargs=extrae_argumentos_general(comando, " ", args, MAX_ARGS,&flagIO ,1,NULL);
            priority=verifyCad(&nargs[0], nargs);
            //printf("\nPriority: %i\n", priority); //See priority to check if works fine
            switch (priority)
            {
                case 0:
                    oneARG(comando);
                case 1:
                    nargsLP= extrae_argumentos_leftPipe(comando, " ", argsLeft, MAX_ARGS, &flagIO);
                    if(flagIO==1){
                        //Caso para 2 comandos establecidos por una pipe ejem: ls | wc
                        fflush(stdout);
                        onePipe(comando,&flagIO);
                    }else if (flagIO == 2){
                        // Caso para 1 comandos con > ejem: ls -la > 1.txt
                        fflush(stdout);
                        redirectOutputToFile(comando,&flagIO);
                    }else if(flagIO == 3){
                        //Caso para comando con < ejemplo wc < 1.txt
                        fflush(stdout);
                        fflush(stdin);
                        redirectInputOfFile(comando,&flagIO);
                    }else if (flagIO == 4){
                        // Caso para 1 comando con >> ejem: ls -la >> 1.txt
                        fflush(stdout);
                        concatredirectOutputToFile(comando,&flagIO);
                    }
                    else if(flagIO == 5){
                        fflush(stdout);
                        fflush(stdin);
                        concatredirectInputOfFile(comando,&flagIO);
                    }
                    break;
                case 2:
                    if (((strcmp(args2[0],"|") == 0) && (strcmp(args2[1],"") == 0)) != 0){
                        fflush(stdout);fflush(stdin);
                        nargsLP=extrae_argumentos_general(comando, " ", argsLeft, MAX_ARGS,&flagIO ,1,NULL);
                        twoPipes(comando,&flagIO);
                    }
                    if (((strcmp(args2[0],"|") == 0) && (strcmp(args2[1],">") == 0)) != 0){
                        fflush(stdout);fflush(stdin);
                        onePipeRedirectOutputToFile(comando,&flagIO);}
                    if (((strcmp(args2[0],"|") == 0) && (strcmp(args2[1],">>") == 0)) != 0){
                        fflush(stdout);fflush(stdin);onePipeconcatredirectOutputToFile(comando,&flagIO);}
                case 3:
                    if (((strcmp(args2[0],"|") == 0) && (strcmp(args2[1],"") == 0) && (strcmp(args2[2],">") == 0)) != 0){
                        fflush(stdout);fflush(stdin);
                        twoPipesRedirectOutputToFile(comando,&flagIO);}
                    if (((strcmp(args2[0],"|") == 0) && (strcmp(args2[1],"") == 0) && (strcmp(args2[2],">>") == 0)) != 0){
                        fflush(stdout);
                        fflush(stdin);
                        twoPipesConcatredirectOutputToFile(comando,&flagIO);}
                    break;
                default:
                    break;
            }
        }
        else{
            close(tuberia[0]);
            printf("\n%s@%s:~%s ", nombre,uname0.nodename, dir);
            leerLinea(&comando[0]);
            write(tuberia[1], comando, strlen(comando)+1);
            wait(&pid);
            close(tuberia[1]);
        }
    }while( strcmp(comando,"exit") != 0);
    return 0;
}

```

Explicando primero se crea una tubería para el manejo de información de entre el proceso padre que pedirá los comandos y el hijo que será el que los ejecute, el hijo hará la comprobación de la flag al pasar por primera vez el comando a verifyCad, ya se explicó el funcionamiento de esta función y con base a la prioridad se sabrá que hacer en el switch case, verify en args 2 almacenará el arreglo de cadenas que tendrán sólo los comandos es decir sólo tendrá “|,>,>>,<<,<<<” para saber después el orden una vez teniendo la prioridad.

Se lee entonces de la tubería el proceso hijo (pid==0) el comando, acá llega y bueno se manda a verificar, se observa la prioridad y se manda al switch este manda a las funciones donde es necesario ejecutar el proceso mientras el padre espera a que termine su hijo, su padre tiene el printf() donde se imprime la información del equipo que ya se guardó, el nombre de la máquina y su dirección para simular la minibash.

Todo esto está en un do while ya que se espera cerrar al leer “exit”.

Conclusion:

Nos costó mucho entender cómo funcionaba la concatenación para archivos pero una vez sabiendo que se tenía que cambiar el modo nos resultó sencillo de implementar, aún no funciona para n procesos pero con una función recursiva quedaría, sólo que por falta de tiempo no se ha implementado además de que muchas funciones quedaron inconclusas pero el programa cumple con los comandos necesarios para la práctica, el uso de dup2 igual nos costó entender pero viendo videos de la clase cuando se explicó pudimos implementarlo.

Referencias:

https://es.wikipedia.org/wiki/Descriptor_de_archivo

<https://manuales.guebs.com/php/function.dio-fcntl.html>

<https://www.thinkage.ca/gcos/expl/c/lib/open.html>

https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Manejo_de_archivos