INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

· CAÑAS SERRANO SERGIO ALEJANDRO

· MARTÍNEZ RODRÍGUEZ ALEJANDRO

2CM7

SISTEMAS OPERATIVOS

PROFESORA ANA BELEM JUÁREZ MÉNDEZ

8 JUNIO DE 2020

PROYECTO

MINISHELL

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, exe, pipe, dup…

Introducción

Un shell es un programa que nos va a permitir acceder a los servicios que nos provee el sistema operativo mediante comandos de textos simples, los cuales nos dan mucha versatilidad a la hora de ejecutar algún programa, modificar archivos; crear carpetas, archivos, etc. facilitandonos la forma en la que podemos acceder a directorios, hacer redireccionamientos de la salida o de entrada así como conocer y manejar con mayor detalle las funcionalidades del sistema operativo.

Además de poder ingresar comandos y que estos sean interpretados por el sistema operativo, los shells ofrecen más elementos que ayudan a mejorar sus funcionalidades, como variables, funciones o estructuras de control. Estos elementos irán cambiando dependiendo del tipo de shell en el que nos encontremos

La shell se basa en una línea de comandos llamada prompt(shells de texto común y shells gráfico común), la cual funcionará como una indicación al usuario para que ingrese un comando, este prompt estará esperando a que el usuario ingrese alguna indicación para que comience a ejecutar dicha indicación..

En los sistemas operativos de Unix existen múltiples implementaciones de shell, los cuales podemos clasificar en dos grandes grupos:

1.- sh (Bourne Shell): este shell fue usado desde las primeras versiones de Unix (Unix Versión 7). Recibe ese nombre por su desarrollador, Stephen Bourne, de los Laboratorios Bell de AT&T. A raíz de él han surgido múltiples shells, tales como zsh (Z shell), ash (almquist shell), bash (Bourne again shell), dash (Debian almquist shell) o ksh (Korn shell).

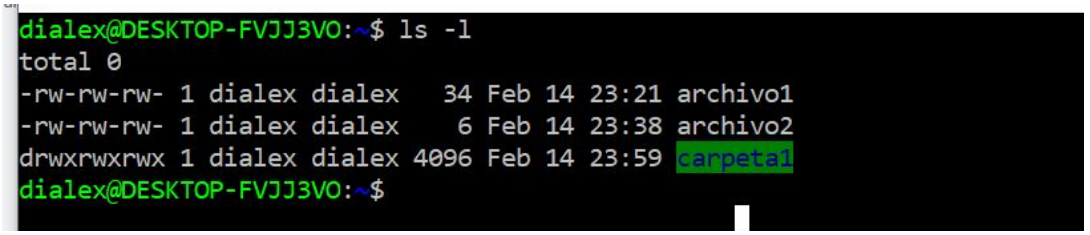
2.-.csh (C shell): caracterizado por presentar una sintaxis muy parecida a la del lenguaje de programación C. Como shell derivados destaca tcsh. Estos shell cuentan con un nivel de uso muy inferior respecto a los de la familia Bourne Shell.

Desarrollo

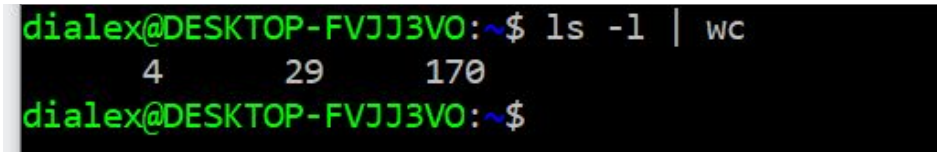
Para la programación del minishell se consideraron 2 casos de entrada:

- La entrada de comandos simples, es decir, aquellos que no hacen uso de tuberías
- La entrada de comandos compuestos, aquellos con una o más tuberías

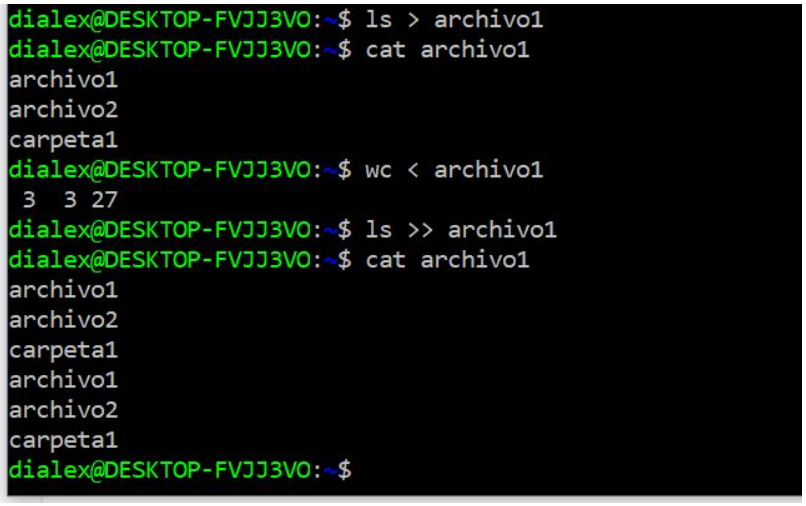
Ejemplo de comando simple:



Ejemplo de comando compuesto:



Se consideró, además, las entradas que tuviesen redireccionamiento del tipo “>”, “>>” y “<”. El tipo de redireccionamiento ingresado se analizaría para el primer y segundo caso en cuanto se recibe la cadena. El tercer tipo de redireccionamiento se analizaría en distintos puntos del programa dependiendo del comando ingresado.



```c
// * delimitador - cadena de referencia para la división de cadenas
// * cadena - la cadena a dividir
// * cads - arreglo de  cadenas que guardará las subcadenas resultantes
int dividirCadena(char* delimitador, char* cadena, char ** cads){
    int x=0;
    char *token, * theRest;
    char *original;

    original=strdup(cadena);
    theRest=original;
    while(token = strtok_r(theRest,delimitador, &theRest)){
            cads[x]=token;
            x++;
    }
    if(strlen(cads[0])==strlen(cadena)){
            return (delimitador == " ") ? 1 : 0; // " " caso especial de división
            //no hubo coincidencias
    }
    else{
            return (delimitador == " ") ? x : 1; // " " caso especial de división
            //hubo coincidencias
    }
}
```

Tanto para las tuberías como para los redireccionamientos se hace uso de una función *dividirCadena* que, con *strtok_r*, realiza la división de cadenas en subcadenas a partir de un *delimitador*, que será la referencia para la separación de cadenas.

La función también servirá para la separación de argumentos de los comandos y la eliminación de espacios.

A partir del hecho de que cada redireccionamiento divide a la entrada en 2 (un comando y un archivo), se busca primero la existencia de un carácter “>” y se verifica en los algoritmos siguientes con la indicación de una bandera de tipo entero *banderaRedir*. Finalmente se cuenta el número de incidencias para definir finalmente el tipo de redireccionamiento.

```c
char* incidencias[]={">", ">>", "<"};//busca las incidencias
int r=0;

if(dividirCadena(incidencias[0], cadena, cads)){//encontro una incidencia de >> o >
    banderaRedir++;//la bandera vale 1 para >>, >
}
r = cuentaMayor(cadena); //número de inicidencias de >
cadena = cads[0];

int i = cuentaPipes(cadena);//cuenta el numero de pipes del comando
```

La forma de buscar y verificar la existencia para un redireccionamiento de tipo “<” es similar al caso previo, considerando la bandera entera *banderaMenorque*.

```c
if(dividirCadena(incidencias[2], cadena, cadenaRe)){//encontro una incidencia de <
    banderaMenorque++;//la bandera de <
}
```

Entonces en *cads* se tiene en la posición 0 la primer parte de la cadena hasta antes de la redirección y en la posición 1 se guardará el nombre del archivo al que redirecciona la salida. En el caso de no haber encontrado alguna incidencia, *cads* en 0 guardará la cadena original.

Posteriormente, se analiza si la cadena ingresada por el usuario es un comando simple o compuesto con el uso de la función *cuentaPipes*, que recibe la cadena y devuelve el número total de incidencias del carácter “|”. Se procede a ejecutar el comando para el caso en donde el resultado sea 0 o >1.

```
int i = cuentaPipes(cadena);//cuenta el numero de pipes del comando

if(i==0){//no hay pipes…
}else{//hay pipes…
}
```

```
int cuentaPipes(char * cadena){
    int i, cont = 0;

    for(i = 0; cadena[i]; i++){
        if(cadena[i] == '|') cont++;
    }

    return cont;
}
```

Para el primer caso, se hace uso de la función salida que recibe las banderas de redireccionamiento y la cadena con los comandos. La función está dirigida al análisis y ejecución de un comando simple, aunque hace uso de una función generalizada para ejecutar las sentencias pertinentes para los caracteres “>”, “>>” y “<”: *redireccionamiento*.

```
void salida(int banderaRedir, int banderaMenorque, char** cadenaRe, char** cads, int r){
    int pid = 0, i, status;
    char * argumentos[200];//comando y argumentos del comando
    int ultimo = dividirCadena(" ", cadenaRe[0], argumentos);//recibe la ultima posicion de argumentos

    argumentos[ultimo] = NULL;//ponemos un nulo al final del vector para exec

    pid = fork();

    if(pid == 0){
        int fi = 0, fo = 0;//ficheros de descriptor de archivos
        redireccionamiento(cadenaRe, cads, banderaMenorque, banderaRedir, fi, fo, r);
        execvp(argumentos[0], argumentos);
    } else{
        wait(&status);
    }
}
```

```
void redireccionamiento(char ** cadenaRe, char** cads, int banderaMenorque, int banderaRedir, int fi, int fo, int r){
    char * sinE[10];
    if(banderaMenorque){//redireccionamos la entrada
        dividirCadena(" ", cadenaRe[1], sinE);//quitamos espacios al archivo
        fi = open(sinE[0], O_RDONLY);//(fichero de entrada, lo abre con permisos de lectura)
        dup2(fi, STDIN_FILENO);//redireccion del fichero a la entrada
        close(fi);
    }
    if(banderaRedir){ //redireccionamos la salida
        dividirCadena(" ", cads[1], sinE);//quitamos espacios al archivo
        int flags[] = {(O_RDWR | O_CREAT | O_TRUNC), (O_RDWR | O_CREAT | O_APPEND)};//(>, >>)
        int banderaCrear = (S_IRUSR | S_IWUSR | S_IXUSR);// si se crea, dar permisos al usario
        fo = open(sinE[0], flags[r-1], banderaCrear);
        dup2(fo, STDOUT_FILENO);//redireccionamiento de la salida
        close(fo);
    }
}
```

La función *salida* divide la cadena del comando en tantos argumentos a considerar en la ejecución, determinados por los espacios en la cadena. Después, procede a crear un procesos hijo con la llamada a *fork(),* que ejecutará el comando recibido. Una vez dentro del proceso hijo, realiza la llamada a *redireccionamiento* que considera las banderas mencionadas anteriormente para definir las salidas y entradas del hijo. Para el caso de redireccionar la salida a un archivo, se consideran las banderas O_TRUNC para “>” y O_APPEND para “>>”; en ambos casos se añaden “O_CREAT” para crear el archivo en caso de que no exista y las banderas S_IWUSR, S_IRUSR y S_IXUSR para permisos de escritura, lectura y ejecución respectivamente para todos los usuarios del sistema. La salida y entrada se asignan con la llamada *dup2*, que duplica el fichero abierto en el que corresponda (entrada estándar o salida estándar).

La ejecución de comandos compuestos inicia dividiendo la cadena principal en subcadenas separadas por “|”. Continúa creando las tuberías con la llamada *pipe* de acuerdo con el número de incidencias del carácter obtenido previamente.

```
dividirCadena("|", cadena, cadenaPipes);
if(dividirCadena(incidencias[2], cadenaPipes[0], cadenaRe)){//encontro una incidencia de <
    banderaMenorque++;//la bandera de <
}

int tuberias[i][2];

for(x = 0; x < i; x++){
    pipe(tuberias[x]);
}
```

Se accede entonces a un ciclo *do - While* que se ejecutará tantas veces resten comandos por realizar (número de divisiones hechas por “|”). En cada inicio del ciclo, se analizará la subcadena correspondiente en busca de argumentos. Finalmente, se inicia la llamada a *fork* donde el proceso hijo y el padre se ejecutará bajo las condiciones siguientes:

- La subcadena es la primera en la cadena original
- La subcadena es la última en la cadena original
- La subcadena se encuentra en medio de la cadena original

En el primer caso se procederá a abrir el extremo de escritura de la primer tubería; en el segundo caso se abrirá el extremo de lectura de la última tubería; y el tercer caso abrirá el extremo de lectura de la tubería que le preceda y el extremo de escritura de la siguiente tubería. En la sección del padre se esperará a que el hijo en ejecución se cierre y, según sea caso, cerrará las tuberías correspondientes. Los casos de redireccionamiento se consideran en la primer y última subcadena. En cada situación, mediante un contador, se ejecuta con *execvp* el comando en turno y sus argumentos.

```
do{
    char * argumentos[200];//comando y argumentos del comando
    int ultimo;
    if(v == 1){
        ultimo = dividirCadena(" ", cadenaRe[0], argumentos);//recibe la ultima posicion de argumentos
        argumentos[ultimo] = NULL;//ponemos un nulo al final del vector para exec
    } else{
        ultimo = dividirCadena(" ", cadenaPipes[v-1], argumentos);//recibe la ultima posicion de argumentos
        argumentos[ultimo] = NULL;//ponemos un nulo al final del vector para exec
    }
```

```
    pid=fork();

    if(pid==0){
        if(v==1){//solo escritura, primer tuberia
            int fi = 0, fo = 0;
            close(tuberias[j][READ_END]);
            dup2(tuberias[j][WRITE_END], STDOUT_FILENO);
            close(tuberias[j][WRITE_END]);
            redireccionamiento(cadenaRe, cads, banderaMenorque, 0, fi, fo, r);
            execvp(argumentos[0],argumentos);
        }
        if(v==i+1){//tuberia extreectura
            j--;
            int fi = 0, fo = 0;
                    //close(tuberias[j][WRITE_END]);
            dup2(tuberias[j][READ_END], STDIN_FILENO);
            close(tuberias[j][READ_END]);
            redireccionamiento(cadenaRe, cads, 0, banderaRedir, fi, fo, r);
            execvp(argumentos[0], argumentos);
        }
        else{//tuberia media, lectura y escritura

            dup2(tuberias[j-1][READ_END], STDIN_FILENO);
            close(tuberias[j-1][READ_END]);

            close(tuberias[j][READ_END]);
            dup2(tuberias[j][WRITE_END], STDOUT_FILENO);
            close(tuberias[j][WRITE_END]);
            execvp(argumentos[0], argumentos);
        }
```

```
                }
                else{//Proceso padre
                    wait(&status);
                    if(v==1){//primer tuberia
                        close(tuberias[j][WRITE_END]);
                    }
                    if(v==i+1) {//ULTIMA TUBERIA
                        close(tuberias[j][READ_END]);
                    }
                    else{//segunda tuberia
                        close(tuberias[j-1][READ_END]);
                        close(tuberias[j][WRITE_END]);
                    }
                }
                v++;
                j++;
            }while(v<=i+1);
        }
        banderaRedir = 0;
        banderaMenorque=0;
        cads[0]=NULL;
        cads[1]=NULL;
    }

    return 0;
}
```

Prompt

Para hacerlo utilizamos un programa en bash que pudiera recuperar y redireccionar las datos que necesitábamos hacia un archivo para después poder recuperarlo en nuestro programa y utilizar esta sentencia a modo de prompt, que son: Usuario del equipo, nombre del equipo y directorio actual

```
#!/bin/dash

usuario=$USER
cadena1=' @ '
cadena2=' : '
cadena3=' $'
epacio=' '


echo $usuario $cadena1 "$(hostname)" $cadena2 $esapacio "$(pwd)" $cadena3 > Promp.txt
```

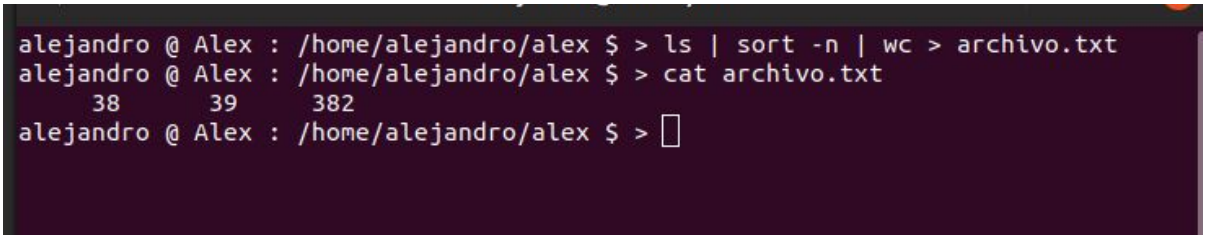
```
if((fp=fopen("Promp.txt", "r"))==NULL){
    printf("No se puede abrir el archivo");
    exit(-1);
}

lectura=getc(fp);

while(lectura!='\n'){
    cadenaPromp[z]=lectura;
    z++;
    lectura=getc(fp);
}

fclose(fp);//cierre del archivo

while(1){

    printf("%s > ", cadenaPromp);
    scanf(" %[^\n]", cadena);//pedimos el comando al usuario
    if(!((strcmp(cadena, "exit"))!=0)){
        exit(-2);
    }
```

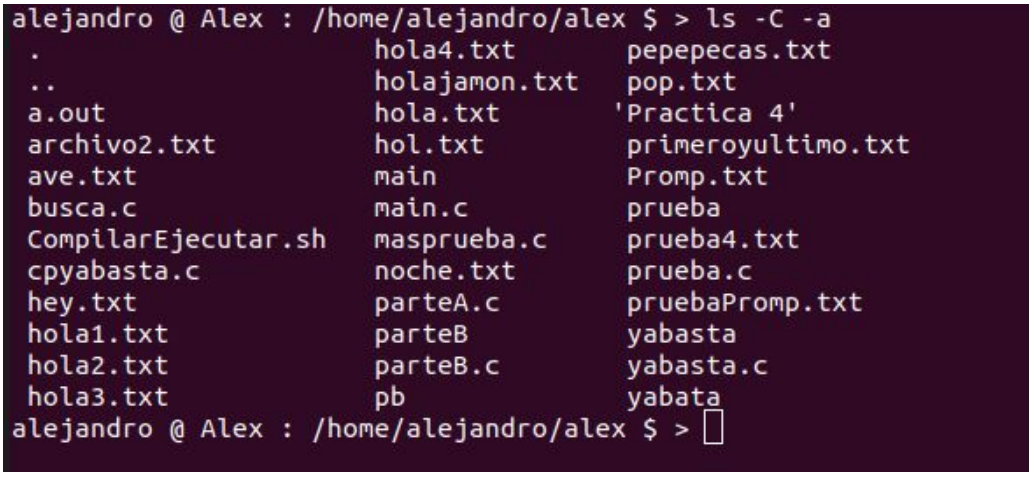
Después en nuestro programa recuperamos el archivo Promp.txt y se lo asignamos a la variable “*cadenaPrompt*” mediante un ciclo while que es el que se encargará de recorrer el archivo con nuestro prompt y a a su vez guardarlo en “*cadenaPrompt*” ” para después

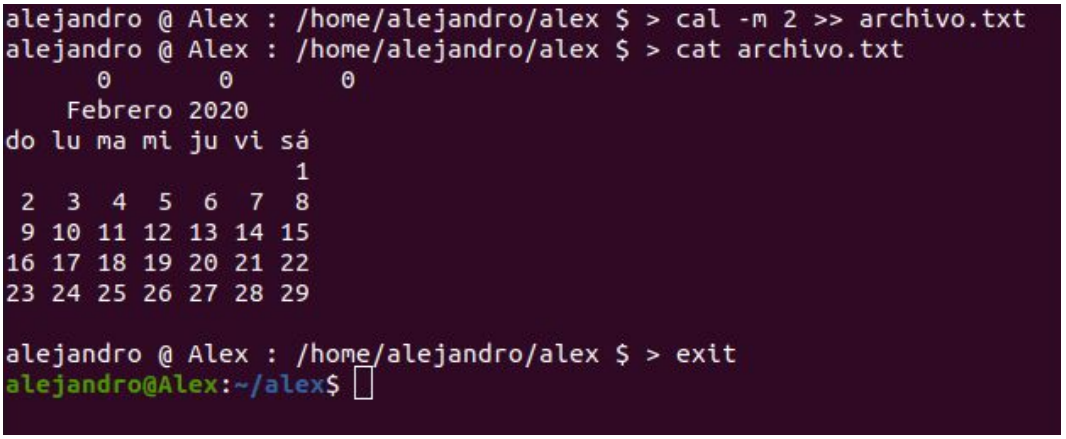
imprimirlo cada vez que se solicite un comando de la terminal al usuario.

Después del cierre del archivo es donde le pedimos al usuario que ingrese el comando , en caso de que este ingresa palabra “exit” el programa termina con su ejecución.

Resultados







Conclusiones

Las herramientas que proporciona C y el entorno de un sistema basado en UNIX aportan un gran número de posibilidades en la programación de aplicaciones y sistemas que adopten el entorno de UNIX. La comunicación y gestión de procesos

es importante a considerar sobre un sistema que requiere la creación de más procesos y las salidas que estos producen.

El seguimiento de flujos de datos es complicado si no se entiende por completo los procesos de comunicación que se utilizan, como lo es el uso de tuberías pipe. Es importante saber dónde y cuándo abrir los extremos de lectura y saber asignarlos los datos correspondientes para que el flujo no se pierda o duplique.

Referencias

1.-Manjarrez, J. (2018). ¿Que es un shell?. Retrieved 8 June 2020, from https://blog.desdelinux.net/que-es-un-shell/

2.-Fernández, F. (2020). Programación shell scrypt en linux. Retrieved 8 June 2020, from http://trajano.us.es/~fjfj/shell/shellscript.htm

3.- Ignacio Lopez, j. (2020). Introducción a la shell de linux. Retrieved 8 June 2020, from http://recursostic.educacion.es/observatorio/web/ca/software/software-general/295-jose-ignacio-lopez

4.-Villagomez, C. (2020). Linux SHELL. Retrieved 8 June 2020, from https://es.ccm.net/contents/316-linux-shell