s

Instituto Politécnico Nacional
Escuela Superior de Computo

Sistemas operativos (2CM9)

Maestra: Ana Belem Juárez Méndez

Proyecto: Mini Bash

Alumno: García González Aarón Antonio

Noviembre 26, 2019

Índice

Objetivo

Realizar un programa mini Bash, el cual muestre los conocimientos adquiridos a lo largo del curso y así como su aplicación en conjunto de varios temas que se vieron por separado a lo largo del curso, como lo es procesos, las funciones exec, tuberías y las funciones dup, entre otros.

Introducción

Bash

Bash (Bourne-again shell) es un programa informático, cuya función consiste en interpretar órdenes, y un lenguaje de consola. Es una shell de Unix compatible con POSIX y el intérprete de comandos por defecto en la mayoría de las distribuciones GNU/Linux, además de macOS. También se ha llevado a otros sistemas como Windows y Android.

Redirecciones de entrada/salida

La sintaxis de Bash permite diferentes formas de redirección de entrada/salida de las que la shell Bourne tradicional carece. Bash puede redirigir la salida estándar y los flujos de error estándar a la vez utilizando la sintaxis.

Por ejemplo, para redireccionar la salida de un comando y volcarla a un archivo bastaría con ejecutar:

$ ls -la ~ > archivo.txt

Podemos, por ejemplo, contar las líneas que tiene un archivo redireccionando la entrada estándar de wc hacia un archivo de texto. Así:

$ wc < archivo.txt

Descriptores de archivos

los términos descriptor de archivo o descriptor de fichero son usados generalmente en sistemas operativos POSIX. En la terminología de Microsoft Windows y en el contexto de la biblioteca stdio, se prefiere el término "manipulador de archivos" o "manipulador de ficheros", ya que es técnicamente un objeto diferente.

En POSIX, un descriptor de archivo es un entero, específicamente del tipo int de C. Hay 3 descriptores de archivo estándar de POSIX que presumiblemente tiene cada proceso:

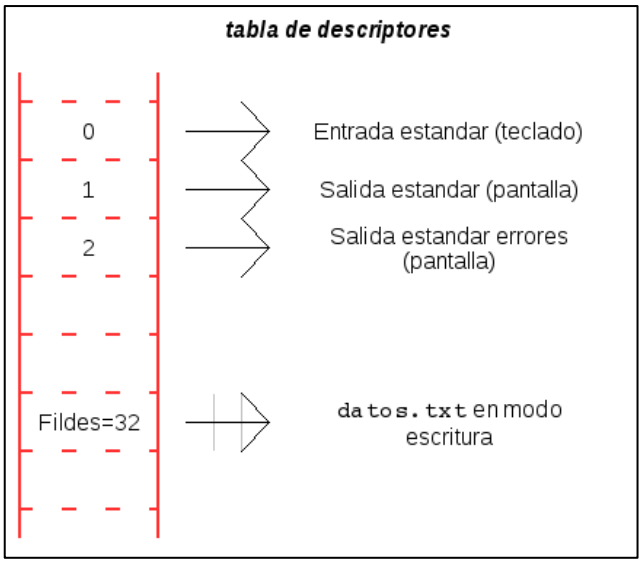
Valor entero	Nombre
0	Entrada estándar (stdin)
1	Salida estándar (stdout)
2	Error estándar (stderr)

Funcion Dup

Las llamadas al sistema dup proporcionan un modo para duplicar un descriptor de archivos, presentando dos o más descriptores diferentes que acceden al mismo archivo. Se pueden usar para leer y escribir en diferentes partes del archivo. La llamada al sistema dup duplica un descriptor de archivo, fildes, enviando un nuevo descriptor. La llamada al sistema dup2 copia eficazmente un descriptor de archivo a otro especificando qué descriptor usar para la copia.

¿Cómo funciona?

Supongamos la siguiente tabla de descriptores de archivos:

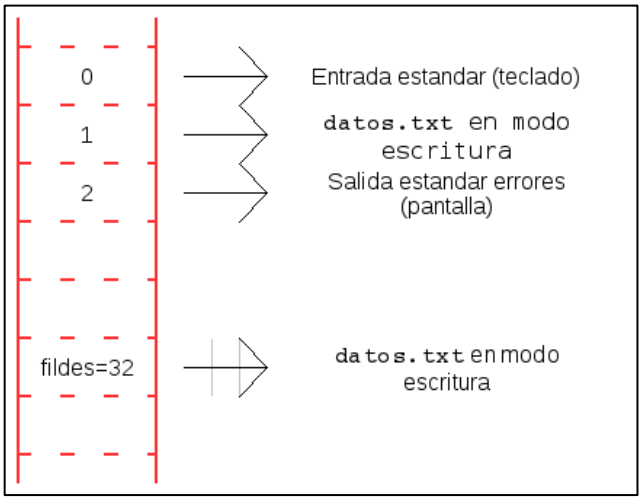


Fíjese que ya existe un descriptor fildes para manejar el fichero datos.txt, que se obtuvo mediante la llamada open, por tanto, el valor del descriptor es seleccionado por el sistema operativo (en nuestro caso, hemos seleccionado el valor 32 arbitrariamente).

A partir de dicha situación, tras realizar la siguiente llamada:

dup2(fd, STDOUT_FILENO);

El resultado sobre la tabla de descriptores es la siguiente:

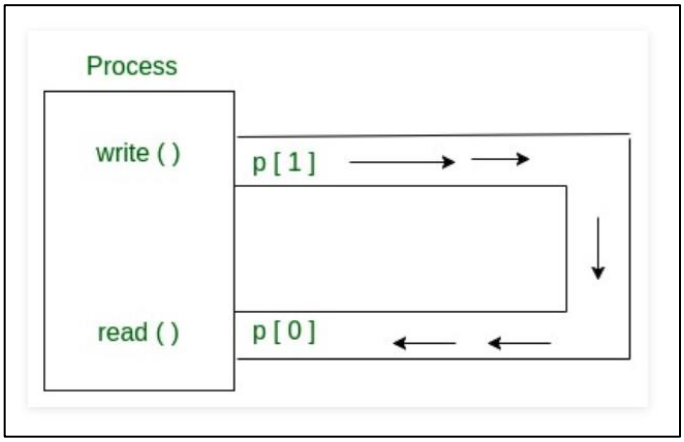


Por tanto, tras la llamada dup2 se genera el duplicado, de manera que el descriptor 1 (STDOUT_FILENO) ya no apunta a la pantalla sino al fichero datos.txt:

De esta manera, cuando invoquemos a printf el mensaje que se pase como parámetro no se imprimirá en pantalla, sino que se almacenará en el fichero datos.txt.

Tuberias

Conceptualmente, una tubería es una conexión entre dos procesos, de modo que la salida estándar de un proceso se convierte en la entrada estándar del otro proceso. En el sistema operativo UNIX, las tuberías son útiles para la comunicación entre procesos relacionados (comunicación entre procesos).



Señales

Una señal es un "aviso" que puede enviar un proceso a otro proceso. El sistema operativo unix se encarga de que el proceso que recibe la señal la trate inmediatamente. De hecho, termina la línea de código que esté ejecutando y salta a la función de tratamiento de señales adecuada. Cuando termina de ejecutar esa función de tratamiento de señales, continua con la ejecución en la línea de código donde lo había dibujado.

El sistema operativo envía señales a los procesos en determinadas circunstancias. Por ejemplo, si en el programa que se está ejecutando en una shell nosotros apretamos Ctrl-C, se está enviando una señal de terminación al proceso. Este la trata inmediatamente y sale. Si nuestro programa intenta acceder a una memoria no válida (por ejemplo, accediendo al contenido de un puntero a NULL), el sistema operativo detecta esta circunstancia y le envía una señal de terminación inmediata, con lo que el programa "se cae".

Desarrollo

Partimos de la función main, de ahí iremos explicando el funcionamiento del programa:

1. Primero declaramos la señal SININT, que estará a la espera a que el usuario teclee Control + C, esto con el propósito de finalizar la ejecución de este, que a su vez llama a la función sigint_handler que se encarga de hacerlo.
2. Declaramos una segunda señal que estará a la espera de la finalización de los hijos del proceso actual (main), esto con la función child_exit, que funciona de la siguiente manera
   2.1. Espera a cualquier proceso hijo
   2.2. Con WNOHANG regresa de inmediato si ningún hijo ha terminado
   2.3. Si no hubo ningún error nos regresa el pid del proceso que termino en pantalla
3. Entra en un bucle infinito para la ejecución de lo que el usuario teclee en la terminal
4. Lee la línea que el usuario tecleo, esta deberá de ser un tamaño definido como máximo
5. Con la función strchr buscamos el apuntador a '|' si lo encuentra entra a un ciclo while que se repetirá tantas veces como '|' tuberías contenga la línea, si hay por lo menos una tubería, lo que ocurre es lo siguiente:
   5.1. Al '|' lo cambia por un '\0' final de cadena, llama a la función run y la función run, llama a la función parse, cuyos funcionamientos son los siguientes:

   La función parse:
   1. Se salta el primer espacio en blanco que encuentre del comando
   2. Con la función strchr buscamos el apuntador a ' ', en donde si mínimo hay un ' ', entrara a un ciclo while que se repetirá tantas veces como ' ' haya en el comando, mete en myargv cada componente del comando, sin espacios en blanco, donde el ultimo apuntador a myargv será nulo (esto con el propositito de identificar que se han terminado de leer todos los subcomandos de la línea oriiginal)
   3. Mete en myarg los apuntadores de cada subcomando del comando
   4. Pone las banderas isInDir, isOutDir en sus valores respectivos para indicar que se encontraron redireccionamientos en el comando y en que dirección ><

   La función run:
   1. Si el comando no es vacío, continua la ejecución del comando
   2. Comprueba si es comando "exit", "help", "cd" u otro comando

3. Si es otro comando, entonces vamos a la parte interesante del programa
    3.1. Llama a la función command, misma que se encargara del trabajo fuerte de este programa (es la función que realmente ejecuta el comando), pero esta función ya sabe si hay o no tuberías, redireccionamientos o ejecución en segundo plano (gracias a las banderas encendidas por parse)
        3.1.1. Por cada llamada a esta función se creará un proceso
        3.1.2. Declaramos la tubería que usaremos [leer, escribir]
        3.1.3. Si se creo correctamente el proceso continuamos
        3.1.4. Identifica si el comando es el primero de todo el comando
            3.1.4.1. Verifica si hay algún redireccionamiento (><), si lo hay verifica cual de los dos redireccionamientos es:
                3.1.4.1.1. Se trata de redireccionamiento (<), lo que ocurre es que obtiene datos a partir de un archivo
                    3.1.4.1.1.1. Duplicamos el descriptor de archivo de la tubería en escritura
                    3.1.4.1.1.2. Cerramos la entrada estándar, por naturaleza de pipe, ya que anteriormente se duplico el descriptor de escritura, es decir se va a escribir y no a leer
                    3.1.4.1.1.3. Abrimos myarg en una posición mas a la derecha
                    3.1.4.1.1.4. Ponemos en NULL a “<” en decir en la posición
                3.1.4.1.2. Se trata de redireccionamiento (>), lo que ocurre es que manda los resultados de un comando a un archivo, que siempre será la salida estándar
                    3.1.4.1.2.1. Duplicamos el descriptor de archivo de la tubería en escritura
                    3.1.4.1.2.2. Cerramos la entrada estándar, por naturaleza de pipe, ya que anteriormente se duplico el descriptor de escritura, es decir se va a escribir y no a leer
                    3.1.4.1.2.3. Abrimos myarg en una posición mas a la izquierda
                    3.1.4.1.2.4. Ponemos en NULL a “>” en decir en la posición
            3.1.4.2. Si no hay redireccionamientos, entonces solo duplicamos el descriptor de archivo de salida y cerramos la entrada
        3.1.5. Si no es primer ni ultimo subcomando, entonces duplica el descriptor de entrada y salida
        3.1.6. Si es el ultimo subcomando
            3.1.6.1. Verifica si hay algún redireccionamiento (><), si lo hay verifica cual de los dos redireccionamientos es:
                3.1.6.1.1. Se trata de redireccionamiento (<), lo que ocurre es que obtiene datos a partir de un archivo
                    3.1.6.1.1.1. Duplicamos el descriptor de archivo de la tubería en escritura
                    3.1.6.1.1.2. Cerramos la entrada estándar, por naturaleza de pipe, ya que anteriormente se duplico el descriptor de escritura, es decir se va a escribir y no a leer
                    3.1.6.1.1.3. Abrimos myarg en una posición mas a la derecha
                    3.1.6.1.1.4. Ponemos en NULL a “<” en decir en la posición
            3.1.6.2. Si no hubo redireccionamiento, entonces duplica el descriptor de archivo de entrada

3.1.7. Ejecuta el comando de myarg, que básicamente regresara un valor que indicara la nueva entrada para el siguiente comando

4. Ejecuta el ultimo subcomando y verifica que todo el comando haya finalizado, esto para que el siguiento comando o command line pueda iniciar limpio

Resultados

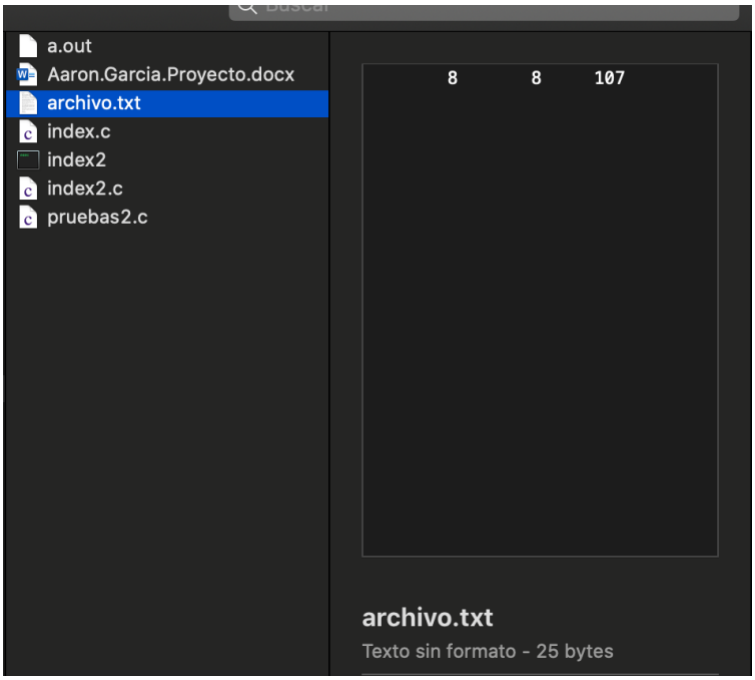
```
[MacBook-Pro-de-Aaron:proyecto aarongarcia$ gcc index.c
[MacBook-Pro-de-Aaron:proyecto aarongarcia$ ./a.out
Teclee [help] para obtener ayuda sobre cómo usar el shell
Aaron-mini-bash>
```

```
Aaron-mini-bash> date
Sun Nov 24 14:39:33 CST 2019
Processo 49022 terminado.
Aaron-mini-bash> cal
   November 2019
Su Mo Tu We Th Fr 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 30

Processo 49023 terminado.
Aaron-mini-bash> ls
Aaron.Garcia.Proyecto.docx      index.c                         pruebas2.c
a.out                           index2                          ~$ron.Garcia.Proyecto.docx
archivo.txt                     index2.c
Processo 49024 terminado.
Aaron-mini-bash>
```

```
Processo 49024 terminado.
Aaron-mini-bash> cal -m 5
      May 2019
Su Mo Tu We Th Fr 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 30 31

Processo 49027 terminado.
Aaron-mini-bash> ls -l -a
total 1936
drwxr-xr-x  11 aarongarcia  staff     352 Nov 24 14:38 .
drwxr-xr-x@ 18 aarongarcia  staff     576 Nov 17 11:46 ..
-rw-r--r--@  1 aarongarcia  staff    6148 Nov 24 12:42 .DS_Store
-rw-r--r--@  1 aarongarcia  staff  899577 Nov 24 14:07 Aaron.Garcia.Proyecto.docx
-rwxr-xr-x   1 aarongarcia  staff   18068 Nov 24 14:38 a.out
-rw-r--r--   1 aarongarcia  staff      25 Nov 24 13:05 archivo.txt
-rw-r--r--   1 aarongarcia  staff    8151 Nov 24 14:08 index.c
-rwxr-xr-x   1 aarongarcia  staff   18076 Nov 24 11:28 index2
-rw-r--r--   1 aarongarcia  staff    8659 Nov 24 12:41 index2.c
-rwxr-xr-x   1 aarongarcia  staff   10671 Nov 24 11:51 pruebas2.c
-rw-r--r--   1 aarongarcia  staff     162 Nov 18 08:41 ~$ron.Garcia.Proyecto.docx
Processo 49028 terminado.
Aaron-mini-bash>
```

```
Aaron-mini-bash> ls | wc | wc
Processo 49048 terminado.
Processo 49049 terminado.
       1       3      25
Processo 49050 terminado.
Aaron-mini-bash>
```

```
Processo 49132 terminado.
Aaron-mini-bash>  ls | sort -n | wc > archivo.txt
Processo 49133 terminado.
Processo 49134 terminado.
Processo 49135 terminado.
Aaron-mini-bash>
```



Conclusiones

Al principio del desarrollo del proyecto no tenia una idea muy clara de como comenzar, empecé por lo básico, el bucle infinito y las funciones de salida, también indague sobre señales y use la sigint, luego pensé en como hacer que las tuberías se conectaran, pensé en hacerlo recursivamente, pero no lo logre, luego indague un poco mas en la red y con la función dup2 y algunas banderas lo logre, debo admitir que la función dup2 me costo un poco de trabajo para entender su funcionamiento, ya que en las paginas en donde la investigue se referían a esta función de manera abstracta, así que busque un ejemplo y con eso fue suficiente, no sabia exactamente la razón de esta función, lo que hace es crear un duplicado en algún descriptor de archivo (0,1 o 2) de otro descriptor de archivo que normalmente se obtiene con la función fopen, pero en este caso aproveche los pipes para poder realizar esto y así cumplir con el objetivo.

Anexo(código)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

// Variables globales
```

```c
#define READ  0
#define WRITE 1
#define MAX_LIMIT 80

static int run(char* cmd, int input, int first, int last);
static void parse(char* cmd);
static int command(int input, int first, int last);
static char* skipwhite(char* s);
void sigint_handler(int sig);
void child_exit();
static int command(int input, int first, int last);
static void waitEnd();

static char* myargv[MAX_LIMIT];
pid_t pid;
int command_pipe[2]; // Para operaciones de tuberías.
int isInDir = 0; // <
int isOutDir = 0; // >
int rPos, lPos; // x ><
static char line[80];// el comando de entrada
static int fgProc = 0;

int main(int argc, char const *argv[]){
   printf("Teclee [help] para obtener ayuda sobre cómo usar el shell\n");
   signal(SIGINT, sigint_handler); // controlador de señal para Ctrl + C
   signal (SIGCHLD, child_exit); // Envia la señal de que su proceso hijo ha terminado y ejecuta la funcion
   // SIGCHLD proceso hijo termiando o parado
   while (1) { // Bucle infinito del programa
      printf("Aaron-mini-bash> "); /* Mostrar el command prompt */
      fflush(NULL);
      if (!fgets(line, MAX_LIMIT, stdin)) // si la linea es valida, se lee
         return 0;

      isOutDir = 0;
      rPos = -1;
      isInDir = 0;
      lPos = -1;
      int input = 0;
```

```
        int first = 1;
        char* cmd = line; // copiamos nuestra linea en cmd
        char* next = strchr(cmd, '|'); // Localiza la primera aparición de |, como apuntador
        while (next != NULL){ // entra si por lo menos hay un '|' en el cmd
            /* 'next' apunta a '|' */
            *next = '\0'; // cambiamos el valor de | a espacio
            input = run(cmd, input, first, 0); // input -> stdin, primer caso , 0 -> bandera que dice que no es el ultimo dentro de la tuberia
            cmd = next + 1;
            next = strchr(cmd, '|'); /* Encontrar siguiente '|' */
            first = 0; // bandera que indica que ya se ha leido por lo menos una instruccion
        }
        input = run(cmd, input, first, 1);
        waitEnd(); // espere a que todo el proceso en primer plano complete la ejecución.
        fgProc=0; // En este punto, todo el proceso de primer plano se ha completado, debido a nuestra llamada de espera.
    }
    return 0;
}

static int run(char* cmd, int input, int first, int last){
    int toReturn, count = 0;
    parse(cmd);
    if (myargv[0] != NULL) { // si no es ultimo comando (comando)
        if (strcmp(myargv[0], "exit") == 0){
            exit(0);
        }
        if (strcmp(myargv[0], "help") == 0){
            printf("Welcome to mini shell!\n");
            printf("The shell supports standard unix/ linux shell commands\n");
            printf("To execute a command use the following syntax\n >command [args]* [ | commmand [args]* ]* \n ex. \nmini-shell>ls -l |
wc\n\n");
            printf("Certain inbuilt commands:\n");
            printf("cd : changes the directory to the said directory, provided it exists.\n Syntax cd <directory path> \n ex. \nmini-shell>cd
newDir\n\n");
            printf("exit : Exits from mini-shell. You can do the same using the Ctrl+c signal.\n ex. \nmini-shell>exit\n\n");
            printf("last [n] : prints the last n commands entered into the shell. If called without n, prints all the commands executed till that
point in time.\n");
            printf("ex: last 4 -> to print the last 4 commands OR last -> to print all the commands that were entered.\n");
            return 0;
```

```
        }
        if (strcmp(myargv[0], "cd") == 0){
            if(chdir(myargv[1]) != 0){ // chdir - cambia el directorio de trabajo
                // Al completar con éxito, se devolverá 0. De lo contrario, se devolverá -1
                printf("%s : No existe ese directorio\n", myargv[1]);
            }
            return 0;
        }else{ // otro comando
            fgProc+=1;
            return command(input, first, last);
        }
    }
    return 0;
}

static char* skipwhite(char* s){
    while (isspace(*s)) ++s;
    return s;
}

static void parse(char* cmd){
    cmd = skipwhite(cmd);
    char* next = strchr(cmd, ' ');
    int i = 0;

    while(next != NULL) { // Minimo hay un ' '
        next[0] = '\0';
        myargv[i] = cmd;
        ++i;
        cmd = skipwhite(next + 1);
        next = strchr(cmd, ' ');
    }

    if (cmd[0] != '\0') { // es decir que no esta vacio
        myargv[i] = cmd;
        next = strchr(cmd, '\n');
        next[0] = '\0';
        ++i;
```

```
    }
    myargv[i] = NULL;

    // verifica > y <
    for(int j=0;myargv[j]!=NULL;j++){
        if(strcmp(myargv[j], "<") == 0){
            isInDir = 1;
            rPos = j;
            break;
        }else{
            isInDir = 0;
            rPos = -1;
        }
    }
    for(int j=0;myargv[j]!=NULL; j++){
        if(strcmp(myargv[j], ">") == 0){
            isOutDir = 1;
            lPos = j;
            break;
        }else{
            isOutDir = 0;
            lPos = -1;
        }
    }
}

// controlador de señal para Ctrl + C
void sigint_handler(int sig){
    printf(" Terminando a través del controlador de señal\n");
    exit(0);
}
// controlador de señal para salida hijo
void child_exit(){
    pid_t cpid;
    while(1){
        cpid = waitpid(-1, NULL, WNOHANG);
        // lo que significa que espera por cualquier proceso hijo; este es el mismo comportamiento que tiene wait.
        // WNOHANG Regrese de inmediato si ningún hijo ha terminado.
```

```
        if (cpid == 0)
            return;
        else if (cpid == -1)
            return;
        else{
            printf ("Processo %d terminado.\n", cpid);
        }
    }
}

static int command(int input, int first, int last){
    FILE* fd;
    int command_pipe[2];
    /* Invocar tubería */
    pipe( command_pipe ); // pipe (int fds [lector,escritor]); -> 0,1
    pid = fork();

    if (pid == 0){ // en el proceso hijo
        if (first == 1 && last == 0 && input == 0) { // primer comando
            if((isInDir) || (isOutDir)){ // si hay < o >
                if(isInDir){ // <, lo que ocurre es que obtiene datos a partir de un archivo
                    dup2(command_pipe[WRITE], STDOUT_FILENO ); // int dup2 (int oldfd, int newfd);
                    // STDOUT_FILENO es un descriptor de archivo entero (en realidad, el entero 1).
                    // Dup2 duplica command_pipe, es decir que manda a la salida estandar lo que se se que tenga la tuberia
                    close(STDIN_FILENO);
                    fd = fopen(myargv[rPos+1],"r"); // se va a leer
                    myargv[rPos] = NULL;
                }else{  // >, lo que ocurre es que manda los resultados de un comando a un archivo, que siempre sera la salida estandar
                    dup2( command_pipe[WRITE], STDOUT_FILENO );
                    close(STDOUT_FILENO);
                    fd = fopen(myargv[lPos+1],"w"); // se va abrir el descriptor de archivo de salida que sera el lPos
                    myargv[lPos] = NULL;
                }
            }else{
                dup2( command_pipe[WRITE], STDOUT_FILENO );  // la entrada sera la entrada entandar en duplicidad de descriptor
            }
        } else if (first == 0 && last == 0 && input != 0) { // Comandos medios, es decir no es el primero ni el ultimo
            dup2(input, STDIN_FILENO);
```

```
            dup2(command_pipe[WRITE], STDOUT_FILENO); // ahora en la duplicidad de c_p[write] se escribira la salida estandar
        }else{ // Ultimo comando
            if((isInDir) || (isOutDir)){
                if(isInDir){
                    dup2( input, STDIN_FILENO);
                    close(STDIN_FILENO);
                    fd = fopen(myargv[rPos+1],"r");
                    myargv[rPos] = NULL;
                }else{
                    dup2(input, STDIN_FILENO);
                    close(STDOUT_FILENO);
                    fd = fopen(myargv[lPos+1],"w");
                    myargv[lPos] = NULL;
                }
            }else{
                dup2( input, STDIN_FILENO ); // ahora en la duplicidad de input se leera la entrada estandar
            }
        }

        execvp( myargv[0], myargv); // nombre del archivo a ejecutar, matriz de cadenas de caracteres
        printf("Comando no encontrado: ¿quiso decir algo más?\n");
        exit(0);
    }
    if (input != 0)
        close(input);

    // Nada más necesita ser escrito
    close(command_pipe[WRITE]);

    // Si es el último comando, no necesita leer nada más
    if (last == 1){
        close(command_pipe[READ]);
    }
    return command_pipe[READ];
}

static void waitEnd(){ /*Espera a que se complete todo el proceso de primer plano.*/
    int i;
```

```
    for(int i=0;i<fgProc;i++)
        wait(NULL); // solo espera a que finalice cualquier hijo.
}
```

Referencias

[1]. dup y dup2. (2014, 16 marzo). Recuperado 24 noviembre, 2019, de https://baulderasec.wordpress.com/programacion/programacion-con-linux/3-trabajando-con-los-archivos/acceso-de-bajo-nivel-a-archivos/dup-y-dup2/

[2]. Colaboradores de Wikipedia. (2019, 12 noviembre). signal.h - Wikipedia, la enciclopedia libre. Recuperado 18 noviembre, 2019, de https://es.wikipedia.org/wiki/Signal.h

[3]. dup() and dup2() Linux system call - GeeksforGeeks. (2017, 26 septiembre). Recuperado 19 noviembre, 2019, de https://www.geeksforgeeks.org/dup-dup2-linux-system-call/

[4]. Execute a Program: the execvp() System Call. (s.f.-a). Recuperado 24 noviembre, 2019, de http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/exec.html

[5]. Execute a Program: the execvp() System Call. (s.f.-b). Recuperado 24 noviembre, 2019, de http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/exec.html

[6]. Function Pointer in C - GeeksforGeeks. (2018, 5 septiembre). Recuperado 18 noviembre, 2019, de https://www.geeksforgeeks.org/function-pointer-in-c/

[7]. Gorka Urrutia, G. GK. (2012, 7 diciembre). Procesos en C: Señales (SIGINT). Recuperado 18 noviembre, 2019, de https://nideaderedes.urlansoft.com/2009/10/13/procesos-en-c-senales-sigint/

[8]. Linux Signals – Example C Program to Catch Signals (SIGINT, SIGKILL, SIGSTOP, etc.). (2012, 8 marzo). Recuperado 18 noviembre, 2019, de https://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/

[9]. pipe() System call - GeeksforGeeks. (2019, 12 junio). Recuperado 19 noviembre, 2019, de https://www.geeksforgeeks.org/pipe-system-call/

[10]. Pipes en Linux. (s.f.-a). Recuperado 20 noviembre, 2019, de http://www.reloco.com.ar/linux/prog/pipes.html

[11]. Pipes en Linux. (s.f.-b). Recuperado 20 noviembre, 2019, de http://www.reloco.com.ar/linux/prog/pipes.html

[12]. strdup () – ¿Qué hace en C? - Código de registro. (s.f.). Recuperado 18 noviembre, 2019, de https://codeday.me/es/qa/20181130/4361.html

[13]. The GNU C Library - Signal Handling. (s.f.). Recuperado 18 noviembre, 2019, de https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_24.html

[14]. Ubuntu Manpage: wait, waitpid - espera por el final de un proceso. (s.f.-a). Recuperado 18 noviembre, 2019, de http://manpages.ubuntu.com/manpages/bionic/es/man2/wait.2.html

[15]. Ubuntu Manpage: wait, waitpid - espera por el final de un proceso. (s.f.-b). Recuperado 18 noviembre, 2019, de http://manpages.ubuntu.com/manpages/bionic/es/man2/wait.2.html

[16]. Unix / Linux - Pipes and Filters - Tutorialspoint. (s.f.). Recuperado 18 noviembre, 2019, de https://www.tutorialspoint.com/unix/unix-pipes-filters.htm

[17]. Unix / Linux - Processes Management - Tutorialspoint. (s.f.). Recuperado 18 noviembre, 2019, de https://www.tutorialspoint.com/unix/unix-processes.htm