# Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadores (DISCA) *Universitat Politècnica de València*

fSO

# Práctica 6
# Llamadas UNIX para archivos
# Versión 2.3

## Contenido

# 1. Objetivos

El sistema de archivos Unix presenta una interfaz única para el manejo de dispositivos y archivos. Las llamadas al sistema relacionadas con archivos son de amplio uso en la programación de aplicaciones. La presente práctica se orienta principalmente al uso de las mismas para la comunicación entre procesos y el redireccionamiento de la E/S. En concreto los objetivos de la práctica son:

- Trabajar las llamadas Unix para el manejo de archivos: open, close, read, write, pipe y dup2.
- Estudiar el mecanismo de redireccionamiento de la E/S a archivos regulares y tubos (pipe).
- Comprender el mecanismo de herencia que permite la comunicación entre procesos.

# 2. Manejo de archivos en Unix

El manejo de archivos en UNIX sigue el modelo de sesión. Para trabajar con archivos primero hay que abrirlo con la llamada open(). La función open() devuelve un descriptor de archivo (*file descriptor*), que es un número entero positivo que identificará al archivo en futuras operaciones. Este valor es la posición de la tabla de descriptores del proceso que contiene el puntero a la tabla de aperturas del sistema. Finalmente hay que cerrar el archivo, con la llamada close(), para liberar los recursos asignados al archivo.

Unix utiliza la misma interfaz para trabajar con archivos que con dispositivos de E/S. Los descriptores 0, 1 y 2 están establecidos para los dispositivos de E/S estándar, se heredan de procesos padre a hijos a través del mecanismo de herencia. El descriptor 0 corresponde a la entrada estándar (por defecto el teclado), el descriptor 1 a la salida estándar (por defecto la pantalla) y el descriptor 2 al dispositivo de visualización de errores (la pantalla también). La utilización de descriptores permite al sistema ser mucho más eficiente en el trabajo con archivos que si se utilizase el nombre propuesto por el usuario.

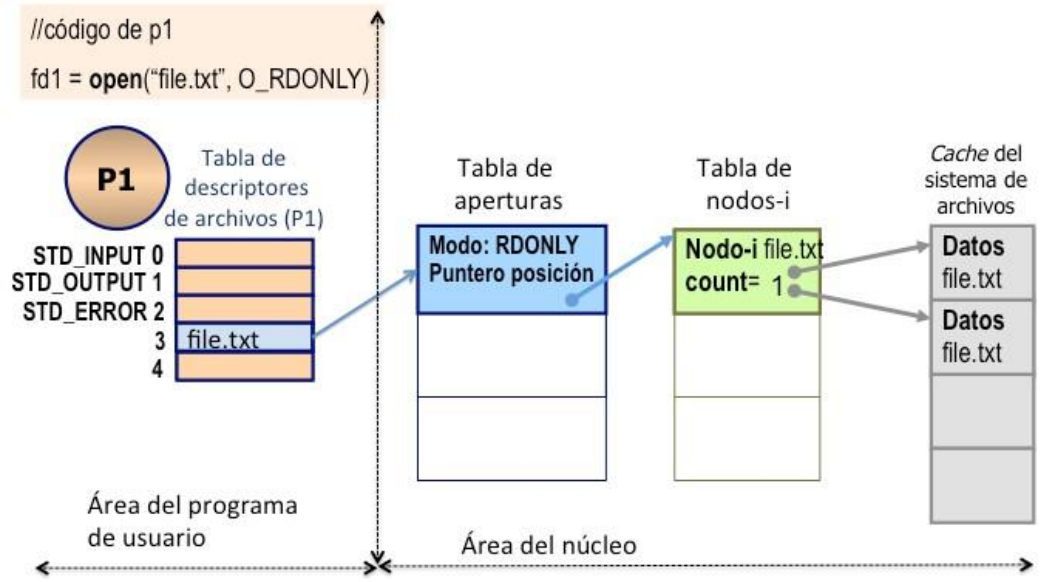


Figura 1: Tabla de descriptores de archivo para un proceso, después de realizar la llamada *open()*

Unix accede a los archivos de forma secuencial, aunque se puede realizar un acceso directo utilizando la llamada lseek(). Cada apertura de archivo dispone de un puntero de posición que se incrementa con cada lectura o escritura un número de bytes igual al número de bytes leídos o escritos. La llamada lseek() permite posicionar el puntero en una determinada posición del archivo.

# 3. Apertura y cierre de archivos

La figura 1 representa gráficamente el efecto de realizar una llamada open(). En Unix los procesos reciben la tabla de descriptores a través del mecanismo de herencia, donde los descriptores 0, 1 y 2 corresponden a la entrada estándar, salida estándar y salida de error (STDIN, STDOUT, STDERR) respectivamente. Unix utiliza la llamada open() para asignar un descriptor de archivo a un archivo o dispositivo físico.

## 3.1 Ejercicio 1: Descriptores de archivo, llamada open()

El contenido del archivo *descriptor.c* proporcionado con el material de prácticas es el siguiente:

```
// descriptor.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc,char *argv[]) {
int fda,fdb;
   if (argc!=2)
{
      fprintf(stderr,"Required read/write file \n");
exit(-1);
  }

  if ((fda=open(argv[1],O_RDONLY))<0)
fprintf(stderr,"Open failed \n");   else
    fprintf(stderr,"Read %s descriptor = %d \n",argv[1],fda);

  if ((fdb=open(argv[1],O_WRONLY))<0)
fprintf(stderr,"Open failed \n");   else
    fprintf(stderr,"Write %s descriptor = %d \n",argv[1],fdb);
return(0);
}
```

Para ejecutar *descriptor.c* ponga como parámetro el nombre del archivo a abrir. Compile y ejecútelo.

```
$gcc descriptor.c –o descriptor

$./descriptor  descriptor.c
```

**Cuestión 1:** Analice el código y el resultado de la ejecución y responda a la siguientes cuestiones:

| |
|---|
| 1. ¿Qué variables corresponden a los descriptores de archivo en el código propuesto?<br><br>Las variables fda y fdb |
| 2. Justifique el número asignado por el sistema a la variable fda<br><br>El numero de la variable fda es la primera posición libre de la tabla de descriptores |
| 3. Justifique el número asignado por el sistema a la variable fdb<br>El numero de la variable fdb es la siguiente a fda ya que esta está ocupada. |

### 3.2 Ejercicio 2: Descriptor de la salida estándar, llamada close()

La llamada close(fd) libera el descriptor fd de la tabla de descriptores. En este ejercicio hay que confirmar que el descriptor de la salida estándar y por tanto de la terminal corresponde al descriptor número 1. Para ello trabaje con el código de *descriptor_output.c* suministrado con el material de prácticas.

```
// descriptor_output.c
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (int argc,char *argv[])
{ char *men1="men1: Writing in descriptor 1 (std_output)\n";
char *men2="men2: Writing in descriptor 2 (std_error)\n";
char *men3="men3: Writing in descriptor 1 (std_output)\n";
char *men4="men4: Writing in descriptor 2 (std_error)\n";
char *men5="men5: Writing in descriptor 1 (std_output)\n";
char *men6="men6: Writing in descriptor 2 (std_error)\n";

  write(1,men1, strlen(men1));
write(2,men2, strlen(men2));
close(1);
  write(1,men3, strlen(men3));
write(2,men4, strlen(men4));
close(2);
  write(1,men5, strlen(men5));
write(2,men6, strlen(men6));
return(0);
}
```

Compile *desriptor_output.c* y ejecútelo.

```
$gcc descriptor_output.c –o descriptor_out

$./descriptor_out
```

**Cuestión 2:** Analice el código y el resultado de la ejecución y responda a las siguientes cuestiones:

1. ¿Qué mensajes se imprimen en la pantalla?

Men1, men2 y men4

2. Justifique por qué no se imprimen cada uno de los mensajes que faltan

Con el método close() liberamos esas posiciones de la tabla de descriptores de archivo. Por ello cuando llegan archivos identificados en esas posiciones liberadas salta un error y no se imprimen.

3. Rellene la tabla de descriptores de archivos abiertos correspondiente a dicho proceso antes del return(0)

| | |
|---|---|
| 0 | std_input |
| 1 | |
| 2 | |
| 3 | |
| 4 | |

# 4. Herencia de descriptores de archivos

Cuando un proceso realiza una llamada fork(), el proceso hijo creado hereda gran cantidad de atributos de su padre, como el directorio de trabajo, atributos de planificación, … y la tabla de descriptores de archivos abiertos. Debido a dicha herencia los procesos comparten el puntero de posición de lectura/escritura de los archivo abiertos antes de la llamada fork().

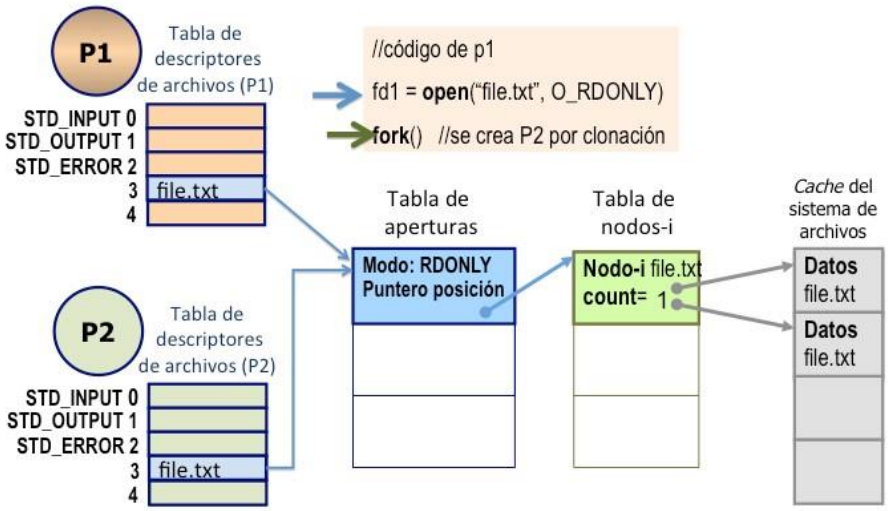


Figura 2: Herencia de la tabla de descriptores de archivos abiertos y su relación con la tabla de aperturas, y las estructuras del sistema.

## 4.1 Ejercicio 3: Proceso padre e hijo comparten archivo

En este ejercicio, el objetivo es estudiar la compartición de archivos por medio de la herencia de descriptores. En concreto, el código de *share_file.c*, que encontrará entre el material de prácticas, utiliza el fichero "messages.txt" para la comunicación entre los procesos padre e hijo.

```c
// share_file.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (int argc,char
*argv[]){    int fd;   pid_t
pid;
  mode_t fd_mode=S_IRWXU;//file permissions
char *parent_message = "parent message \n";
char *child_message = "child message \n";


  fd=open("messages.txt",O_RDWR | O_CREAT,fd_mode);
write(fd,parent_message,strlen(parent_message));


  pid=fork();     if  (pid==0){            write(fd,
child_message,strlen(child_message));
close(fd);      exit(0);
  }   wait(NULL);   write(fd,
parent_message,strlen(parent_message));
close(fd);   return(0);
}
```

Compile *share_file.c* , ejecútelo y muestre el contenido del archivo *mensajes.txt.*

```
$gcc share_file.c –o share

$./share

$ cat messages.txt
```

**Cuestión 3:** Analice el código y el resultado de la ejecución y responda a las siguientes cuestiones:

1. ¿Cuál es el contenido del archivo messages.txt?

Parent message
Child message
Parent message

2. Tanto el proceso padre como el hijo han escrito su mensaje en el archivo messages.txt. ¿Qué mecanismos/llamadas lo han hecho posible?

Al ejecutarse el fork(), el hijo hereda la tabla de descriptores del padre y por tanto es posible que escriban dentro del mismo archivo.

3. Rellene la tabla de descriptores de archivos abiertos correspondiente al proceso padre e hijo antes de ejecutar close(fd);

| | |
|---|---|
| 0 | STD_INPUT |
| 1 | STD_OUTPUT |
| 2 | STD_ERROR |
| 3 | Messages.txt |
| 4 | |

| | |
|---|---|
| 0 | STD_INPUT |
| 1 | STD_OUTPUT |
| 2 | STD_ERROR |
| 3 | Messages.txt |
| 4 | |

# 5. Redireccionamiento: llamada dup2

La redirección de la entrada estándar permite a un proceso "leer" datos de un origen distinto del terminal a través del descriptor 0. Por ejemplo:

```
$ mailx fso10 < mensaje
```

El mensaje a enviar por la aplicación mailx se encuentra en el archivo mensaje. La redirección de la salida estándar permite a un proceso "escribir" datos en un destino distinto del terminal, a través del descriptor 1. Por ejemplo:

```
$ echo hola > f1.txt
```

El resultado de la orden echo se escribe en el fichero f1.txt. La redirección de la salida estándar de errores permite a un proceso "escribir" los mensajes de error en un destino distinto del terminal, a través del descriptor 2. Por ejemplo:

```
$ gcc programa1.c –o programa 2 > errores
```

donde los errores de compilación (orden gcc) del fichero programa1.c se escriben en el fichero errores. El redireccionamiento de la entrada, salida o salida de error estándar en Unix se realiza invocando la llamada dup2.

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Dup2 cierra el descriptor newfd y copia el puntero asociado al descriptor oldfd en newfd. En el anexo de esta práctica se describe con detalle la llamada dup2.

## 5.1 Ejercicio 4: Redirección de la salida estándar a archivo

En este ejercicio se practica como utilizar la llamada dup2() para conseguir que todo lo que se escribe sobre la salida estándar sea redireccionado a un archivo. Para ello deberá utilizar el código proporcionado con la práctica en el archivo *redir_output.c* y mostrado a continuación.

```c
// redir_output.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc,char *argv[]) {
int fd;
  char *arch= "output.txt";
  mode_t fd_mode = S_IRWXU;// file permissions

  fd = open(arch,O_RDWR |
O_CREAT,fd_mode);   if
(dup2(fd,STDOUT_FILENO) == -1)      {
printf("Error calling dup2\n");
exit(-1);
    }
  fprintf(stdout,"out: Output
redirected\n");   fprintf(stderr,"error: not
redirected\n");   fprintf(stderr,"Check file
%s\n",arch);   close(fd);   return(0);
}
```

Compile *redir_output.c*, ejecútelo y muestre el contenido del archivo *output.txt.*

```
$gcc redir_output.c -o redir_out

$./redir_out

$more output.txt
```

**Cuestión 4:** Analice el código y el resultado de la ejecución y responda a las siguientes cuestiones:

1. Justifique, utilizando las instrucciones del código, el contenido del fichero output.txt

Con la instrucción dup2 se hace un redireccionamiento de la salida estándar. La salida del terminal se recoge en output.txt

2. Justifique por qué la llamada open() se ha invocado con los flags "O_RDWR|O_CREAT"

Esas flags sirven para darle permiso tanto de lectura como de escritura en el caso de que no la tuviera. En el caso de ya lo tenga, da error.

3. Rellene la tabla de descriptores de archivos abiertos correspondiente al proceso, justo antes del " if(dup…)"

0

| | |
|---|---|
| 1 | STD_INPUT |
| 2 | STD_OUTPUT |
| 3 | Output.txt |

| | |
|---|---|
| 4 | |

4. Rellene la tabla de descriptores de archivos abiertos correspondiente al proceso antes de ejecutar return(0)

| | |
|---|---|
| 0 | |
| 1 | STD_INPUT |
| 2 | Output.txt |
| 3 | STD_ERROR |
| 4 | Output.txt |

### 5.2 Ejercicio 5: Redirección de la salida estándar a archivo

La tabla de descriptores de archivos abiertos de un proceso no cambia al ejecutar la llamada exec(), por lo que el proceso conserva los archivos abiertos y las redirecciones que se hubieran realizado previas al exec(). El objetivo de este ejercicio es escribir un programa denominado *ls_redir.c* que al ejecutarlo ejecute a su vez la orden "ls -la", redireccionando la salida al archivo *ls_salida.txt*. El resultado final debe ser equivalente a la siguiente orden del shell:

```
$ls –la >ls_salida.txt
```

Para ello copie *redir_output.c* en *ls_redir.c.* Edite *ls_redir.c* y añada en él la llamada execl() en el lugar adecuado, asegurándose de que la salida se redirecciona a *ls_salida.txt.*

```
  execl("/bin/ls","ls","-
la",NULL)
```

Compile ls*_redir.c* , ejecútelo y muestre el contenido del archivo *ls_salida.txt.*

```
$gcc ls_redir.c –o ls_redir

$./ls_redir

$more ls_salida.txt
```

**Cuestión 5:** Analice el código y el resultado de la ejecución y responda a la siguiente cuestión:

Tras la ejecución justifique dónde se ha almacenado el contenido del directorio actual.

Debido al redireccionamiento que hemos hecho, ahora el contenido de la instrucción "ls -la" se almacena en el fichero ls_salida.txt

### 5.3 Ejercicio 6: Redirección de la entrada estándar desde archivo

Escriba un programa denominado *cat_redir.c* que al ejecutarlo ejecute la orden cat redireccionando la entrada al fichero *ls_salida.txt*, igual que si fuese la orden del shell:

```
$cat <ls_salida.txt
```

Para ello copie *ls_redir.c* en *cat_redir.c*. Edite *cat_redir.c* y modifíquelo. Asegúrese de que el archivo *ls_salida.txt* sólo se abre para lectura al invocar la llamada open().

Compile cat_*redir.c* y ejecútelo. Asegúrese de que existe el archivo *ls_salida.txt*.

```
$gcc cat_redir.c –o cat_redir

$./cat_redir
```

**Cuestión 6:** Analice el código y el resultado de la ejecución y responda

| ¿Qué ha sido necesario modificar en el código del ejercicio5 para llevar a cabo el ejercicio 6?<br><br>Se ha cambiado la flag de la llamada open() de O_RDWR -> O_RDONLY, y en la llamada dup2 STDOUT_FLIENO -> STDIN_FILENO, y en execl() /bin/ls -> /bin/cat/ y /ls -> cat |
|---|

# 6. Creación de Tubos: pipe()

En Unix los tubos son un mecanismo de comunicación entre procesos. Un tubo es un archivo sin nombre con dos descriptores, uno de lectura y otro de escritura. Estos dos descriptores permiten utilizar punteros de posición de lectura y escritura diferenciados, de manera que el puntero de lectura únicamente avanza cuando se realizan operaciones de lectura y el de escritura cuando se realizan operaciones de escritura. En UNIX la llamada a sistema para crear un tubo es pipe() (ver anexo):

```
int pipe(int fildes[2])
```

La llamada pipe() crea un buffer con un esquema FIFO de gestión del buffer. Con el descriptor fildes[0] se accede para lectura (entrada) y con fildes[1] para escritura (salida). Los tubos no poseen ningún nombre externo, por lo que sólo pueden ser utilizados mediante sus descriptores por el proceso que lo crea y por los procesos hijos que hereden de éste la tabla de descriptores con fork(). Los procesos deben compartir el tubo y redireccionar su entrada o salida al tubo. Por ejemplo:

```
$ ls | grep txt
```

Esta línea de órdenes visualizará por la salida estándar los nombres de archivos del directorio actual que contengan la cadena txt.

## 6.1 Ejercicio 7: Comunicación de dos procesos mediante pipe()

El objetivo de este ejercicio es desarrollar un programa que sea equivalente a la ejecución de la siguiente línea de órdenes en el Shell:

```
$ ls –la | wc –l
```

Como muestra la figura-3 la orden ls debe redireccionar su salida al tubo, mientras que la orden wc debe redireccionar su entrada para leer del tubo y su salida al archivo *salida.txt*.

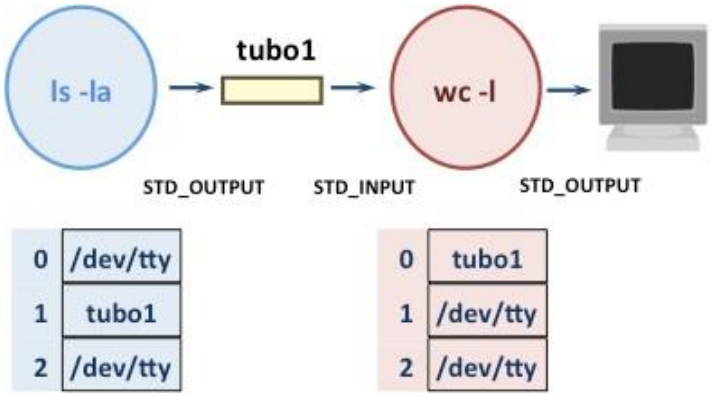


Figura 3: Esquema de redireccionamiento con tubo que hay que implementar en el ejercicio 7.

Por tanto el alumno deberá desarrollar un código donde se realicen las siguientes acciones:

1. El proceso padre crea un tubo
2. Se crea un proceso hijo
    a. Hijo redirecciona al tubo y cierra descriptores
    b. Hijo cambia su imagen de memoria para ejecutar orden ls
3. Se crea otro proceso hijo
    a. Hijo redirecciona del tubo y cierra descriptores
    b. Hijo cambia su imagen de memoria para ejecutar orden wc
4. Proceso padre cierra descriptores y espera a sus hijos.

A modo de código guía se proporciona el archivo *a_pipe.c* el cual contiene líneas de comentarios que el alumno debe reemplazar por instrucciones y llamadas para terminar el ejercicio

```
// a_pipe.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc,char *argv[]) {
int i;
  char* arguments1 [] = { "ls", "-la", 0
};   char* arguments2 [] = { "wc", "-l", 0
};   int fildes[2];   pid_t pid;
  //Parent process creates a pipe
      if ((pipe(fildes)==-1)){
fprintf(stderr," Pipe failure  \n");
exit(-1);
      }
      for (i=0;i<2;i++){
            pid=fork(); //Creates a child process
            if ((pid==0) && (i==0))
            {
              // Child process redirects its output to the pipe
              // Child process closes pipe descriptors

              // Child process changes its memory image
              if (execvp("ls",arguments1)<0){
                fprintf(stderr, "ls not found \n");
exit(-1);
              }
            }
            else if ((pid==0) && (i==1)){

              // Child process redirects its input to the pipe
              // Child process closes pipe descriptors
              // Child process changes its memory image
              if (execvp("wc",arguments2)<0){
                fprintf(stderr, "wc not found \n");
exit(-1);
              }
            }
      }
```

```
      // Parent process closes pipe descriptors
      close(fildes[0]);
      close(fildes[1]);       for
(i=0;i<2;i++) wait(NULL);
      return(0);
}
```

Antes de ejecutar su programa *a_pipe* modificado compruebe cuál es el resultado de las órdenes del shell que trata de implementar. Posteriormente ejecute el programa y compruebe que el resultado es el mismo. Compile *a_pipe.c* , ejecútelo.

```
$gcc a_pipe.c –o a_pipe
$ls –la | wc –l
$./a_pipe
```

**Cuestión 7:** Analice el código y el resultado de la ejecución y responda

| |
|---|
| 1. ¿Qué muestra el proceso en la salida estándar?<br><br>Imprime el numero 20 por pantalla |

### 6.2 Ejercicio 8 (Opcional): Dos tubos con tres procesos

Basándose en el esquema seguido en el ejercicio 7, desarrolle un programa denominado *dos_tubos.c* que ejecute la siguiente línea de órdenes:

```
$ ls –la | grep ejemplo | wc –l > result.txt
```

La estructura de redirecciones necesaria es la que se muestra en la figura 4.

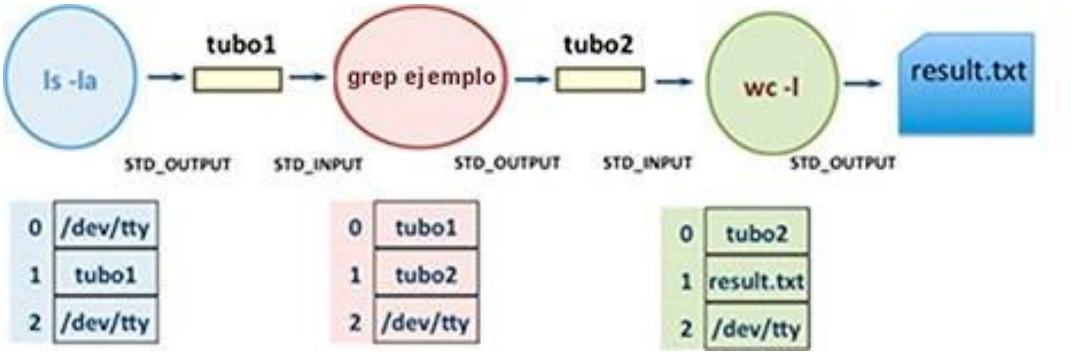


Figura 4: Esquema de redireccionamiento y tubos que hay que implementar en el ejercicio 8

## 7. Anexo: sintaxis de llamadas al sistema

### 7.1 Llamadas open() y close ()

La llamada open() de Unix

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags [,mode_t mode]);
```

Descripción

La llamada open abre el fichero designado por pathname y devuelve el descriptor de fichero asociado.

pathname apunta al nombre de un fichero. flags se utiliza para indicar el modo de apertura del fichero. Dicho modo se construye combinando los siguientes valores:

O_RDONLY: modo lectura.

O_WRONLY: modo escritura.

O_RDWR: modo lectura y escritura.
O_CREAT: si el fichero no existe, lo crea con los permisos indicados en mode.
O_EXCL: si está puesto O_CREAT y el fichero existe, la llamada da error.
O_APPEND: el fichero se abre y el offset apunta al final del mismo. Siempre que se escriba en el fichero se hará al final del mismo.

mode es opcional y permite especificar los permisos que se desea que tenga el fichero en caso de que se esté creando.

Retorno

>0; Retorna un número positivo que corresponde al descriptor del fichero si tiene éxito -1; si hay error.

La llamada close() de Unix

```
#include <unistd.h>
int close(int fd);
```

Descripción

La llamada close libera una posición de la tabla de descriptores de ficheros

Retorno

0 : Retorna 0 si no hay error -1 : si hay algún error.

## 7.2 Llamadas read () y write()

La llamada read() de Unix

```
#include <unistd.h> ssize_t read(int fd, void *buf, size_t count);
```

Descripción

La llamada read lee un número de bytes dado por count del fichero al que hace referencia el descriptor de fichero fd y los coloca a partir de la dirección de memoria apuntada por buf.

Retorno

>0 : si éxito. Retorna el número de bytes leídos
0 : si encuentra el final del fichero -1 : si hay error.

La llamada write() de Unix.

```
#include <unistd.h> ssize_t write(int fd, const void *buf, size_t count);
```

Descripción

La llamada write escribe un número de bytes dado por count en el fichero cuyo file descriptor viene dado por fd. Los bytes a escribir deben encontrarse a partir de la posición de memoria indicada en buf.

Retorno

>0 : si éxito. Retorna el número de bytes escrito
-1 : si hay un error

## 7.3 Llamada pipe()

La llamada pipe() de Unix

```
#include <unistd.h> int
pipe(int fildes[2]);
```

Descripción

Crea un canal de comunicación. El parámetro fildes al retorno contiene dos descriptores de fichero, fildes[0] contiene el descriptor de lectura y fildes[1] el de escritura.

La operación de lectura en fildes[0] accede a los datos escritos mediante fildes[1] como en una cola FIFO (primero en llegar, primero en servirse).

Retorno

0 : si no hay error -1 : si hay algún error.

## 7.4 Llamadas dup y dup2

Las llamadas dup() y dup2() de Unix

```
#include <unistd.h> int dup (int
oldfd);
int dup2(int oldfd, int newfd);
```

Descripción

Duplica un descriptor de fichero.

dup duplica el descriptor oldfd sobre la primera entrada de la tabla de descriptores del proceso que esté vacía. dup2 duplica el descriptor oldfd sobre el descriptor newfd. En el caso en que éste ya hiciera referencia a un fichero, lo cierra antes de duplicar.

Retorno

0 : Ambas devuelven el valor del nuevo descriptor de archivo –1 : en caso de error